Processamento Morfológico

DE IMAGENS DE SENSORIAMENTO REMOTO

GUILHERME PINA CARDIM



Atena Editora
Ano 2024

Processamento Morfológico

DE IMAGENS DE SENSORIAMENTO REMOTO

GUILHERME PINA CARDIM

Atena Editora
Ano 2024

CIÊNCIAS EXATAS E DA TERRA E ENGENHARIAS

Editora chefe
Profª Drª Antonella Carvalho de Oliveira
Editora executiva
Natalia Oliveira
Assistente editorial
Flávia Roberta Barão
Bibliotecária
Janaina Ramos
Projeto gráfico
Camila Alves de Cremo
Ellen Andressa Kubisty
Luiza Alves Batista
Nataly Evilin Gayde
Thamires Camili Gayde
Imagens da capa
iStock
Edição de arte
Luiza Alves Batista

CIÊNCIAS EXATAS E DA TERRA E ENGENHARIAS

Processamento morfológico de imagens de sensoriamento remoto

Diagramação: Ellen Andressa Kubisty
Correção: Yaiddy Paola Martinez
Indexação: Amanda Kelly da Costa Veiga
Revisão: O autor
Autor: Guilherme Pina Cardim

Dados Internacionais de Catalogação na Publicação (CIP)

P963 Processamento morfológico de imagens de sensoriamento remoto / Organizador Guilherme Pina Cardim. – Ponta Grossa - PR: Atena, 2024.

Formato: PDF
Requisitos de sistema: Adobe Acrobat Reader
Modo de acesso: World Wide Web
Inclui bibliografia
ISBN 978-65-258-2311-9
DOI: https://doi.org/10.22533/at.ed.119240703

1. Processamento de imagens digitais. 2. Morfologia matemática. 3. Sensoriamento remoto. I. Cardim, Guilherme Pina (Organizador). II. Título.

CDD 006.696

Elaborado por Bibliotecária Janaina Ramos – CRB-8/9166

Atena Editora
Ponta Grossa – Paraná – Brasil
Telefone: +55 (42) 3323-5493
www.atenaeditora.com.br
contato@atenaeditora.com.br

DECLARAÇÃO DO AUTOR

O autor desta obra: 1. Atesta não possuir qualquer interesse comercial que constitua um conflito de interesses em relação ao conteúdo publicado; 2. Declara que participou ativamente da construção dos respectivos manuscritos, preferencialmente na: a) Concepção do estudo, e/ou aquisição de dados, e/ou análise e interpretação de dados; b) Elaboração do artigo ou revisão com vistas a tornar o material intelectualmente relevante; c) Aprovação final do manuscrito para submissão.; 3. Certifica que o texto publicado está completamente isento de dados e/ou resultados fraudulentos; 4. Confirma a citação e a referência correta de todos os dados e de interpretações de dados de outras pesquisas; 5. Reconhece ter informado todas as fontes de financiamento recebidas para a consecução da pesquisa; 6. Autoriza a edição da obra, que incluem os registros de ficha catalográfica, ISBN, DOI e demais indexadores, projeto visual e criação de capa, diagramação de miolo, assim como lançamento e divulgação da mesma conforme critérios da Atena Editora.

DECLARAÇÃO DA EDITORA

A Atena Editora declara, para os devidos fins de direito, que: 1. A presente publicação constitui apenas transferência temporária dos direitos autorais, direito sobre a publicação, inclusive não constitui responsabilidade solidária na criação dos manuscritos publicados, nos termos previstos na Lei sobre direitos autorais (Lei 9610/98), no art. 184 do Código Penal e no art. 927 do Código Civil; 2. Autoriza e incentiva os autores a assinarem contratos com repositórios institucionais, com fins exclusivos de divulgação da obra, desde que com o devido reconhecimento de autoria e edição e sem qualquer finalidade comercial; 3. Todos os e-book são *open access, desta forma* não os comercializa em seu site, sites parceiros, plataformas de *e-commerce,* ou qualquer outro meio virtual ou físico, portanto, está isenta de repasses de direitos autorais aos autores; 4. Todos os membros do conselho editorial são doutores e vinculados a instituições de ensino superior públicas, conforme recomendação da CAPES para obtenção do Qualis livro; 5. Não cede, comercializa ou autoriza a utilização dos nomes e e-mails dos autores, bem como nenhum outro dado dos mesmos, para qualquer finalidade que não o escopo da divulgação desta obra.

APRESENTAÇÃO

Este livro traz uma fundamentação teórica sobre os principais conceitos que envolvem o processamento morfológico de imagens com foco no processamento de imagens obtidas por sensoriamento remoto. Os conceitos apresentados no livro são parte de uma intensa pesquisa literária condizente com o tema em discussão. Todo conteúdo aqui apresentando é resultado da pesquisa científica e do projeto desenvolvido durante os estudos a nível de mestrado do autor. Com foco em extração de feições a partir de imagens de sensoriamento remoto, além dos conceitos teóricos, o livro apresenta o resultado obtido com o desenvolvimento de uma biblioteca de funções e um sistema computacional livre para processamento morfológico de imagens. Além das tradicionais ferramentas morfológicas, o livro traz uma metodologia semiautomática capaz de identificar feições cartográficas interesse na imagem orbital a partir de amostras da feição de interesse cedidas pelo usuário e uma metodologia automatizada para avaliação dos resultados obtidos por algoritmos de extração.

RESUMO

Pesquisas sobre extração de feições cartográficas de interesse são motivadas, sobretudo, pela crescente importância dos Sistemas de Informações Geográficas e a necessidade de aquisição e atualização de dados espaciais. No campo de planejamento urbano os dados espaciais são utilizados para planejamentos e tomadas de decisão. Para tanto, é imprescindível que estes dados sejam atuais e acurados. Deste modo, uma possibilidade é efetuar a detecção e/ou extração das feições de interesse a partir de imagens de sensoriamento remoto, motivo pelo qual tais estudos são de fundamental importância. Todavia, o que dificulta este procedimento é o conteúdo das imagens envolvidas, o que torna a extração de feições de interesse um tópico desafiador. Além disso, normalmente os sistemas computacionais utilizados para realizar estudos de extração de feições cartográficas são sistemas de domínio particular, tendo suas funcionalidades fechadas, impossibilitando o estudo, alterações e melhorias nesses algoritmos. Neste sentido, este trabalho consiste no desenvolvimento de um sistema computacional, denominado CARTOMORPH, para o processamento de imagens de sensoriamento remoto. Este sistema, implementado de forma livre e de domínio público, tem como finalidade possibilitar a utilização das funções implementadas, focadas no estudo de extração de feições por morfologia matemática, e permitir que alterações, adaptações e melhorias sejam feitas sempre que necessário. Para tanto, foram utilizadas técnicas de processamento digital de imagens, focado em morfologia matemática, para que seja possível o desenvolvimento de rotinas capazes de extrair feições cartográficas de interesse presentes em uma imagem digital. A implementação do sistema CARTOMORPH é de fundamental importância, uma vez que ele é focado em estudos cartográficos para detecção e/ou extração de feições presentes em imagens de sensoriamento remoto. O sistema ameniza dificuldades encontradas por pesquisadores da área, ao utilizar os sistemas comerciais existentes, por possuir funcionalidades dedicadas em estudos de extração de feições cartográficas. Além disso, o código fonte do sistema é disponibilizado, o que permite alterações, bem como a proposição de novos algoritmos, o que não acontece no sistema utilizado atualmente na FCT/Unesp e por grande parte da comunidade científica. A metodologia empregada neste trabalho fundamenta-se em modelos matemáticos e técnicas de processamento digital de imagens para viabilizar a implementação de rotinas de extração de feições cartográficas. Dessa forma, o sistema possibilita que usuários apliquem as operações necessárias durante o desenvolvimento e aplicação de novas metodologias de extração de feições cartográficas. Testes de todas as funcionalidades implementadas foram realizados garantindo a eficiência das operações. Além disso, o sistema foi utilizado para o desenvolvimento de uma metodologia de extração de feições cartográficas, a qual obteve valores de correspondência acima de 90%, em praticamente todas as imagens utilizadas, além de ter sido executada com tempo inferior do que ao utilizar-se de outros sistemas. O CARTOMORPH é disponibilizado para a comunidade científica, possibilitando que este seja livremente utilizado por pesquisadores e estudiosos da área.

PALAVRAS-CHAVE: Processamento digital de imagens; morfologia matemática; CARTOMORPH; extração de feições cartográficas; sensoriamento remoto.

ABSTRACT

Nowadays, the increasing importance of the Geographic Information Systems and the necessity of acquisition and update of spatial data motivate several research about cartographic features extraction. In the urban planning field, the spatial data are used to planning and decision-making. Therefore, it is essential that these data are updated and accurate. Thus, to detect cartographic features using remote sensing images is a significant possibility, reason that these studies are of fundamental importance. Nevertheless, the content of the images involved complicates this procedure and makes it a challenging topic. Moreover, the software used for research about cartographic features extraction methodologies are, usually, of private domain and consequently have the functionalities blocked, disallowing changings and improvement of the algorithms. This sense, this work consists of a software development, named as CARTOMORPH, to remote sensing image processing. The software purpose is to allow the use of the functions implemented, as well as allow changings, adaptions and improving of the functions, since it is of public domain. So, techniques of digital image processing, focused on the mathematical morphology theory, was implemented to enable the development of cartographic features extraction routines. The CARTOMORPH development is of fundamental importance since it is focused on cartographic studies to detect interest features from remote sensing images. The developed system eases the difficulties encountered by researchers because it has dedicated functionalities for cartographic features detection studies. In addition, the code is available to make changings and to propose new functionalities. The methodology applied is based on mathematical models and digital image processing techniques to enable the development of cartographic features detection routines. This way, the software developed allows the users to apply the necessary operations during the development of new methodologies for cartographic features extraction. Tests of all functionalities implemented were performed by the comparison of the results with mathematical calculations and results of others software consolidated. Furthermore, the system was used to develop a semiautomatic methodology to detect cartographic features, which achieve statistical values over 90% for most images tested and the time necessary to perform it was smaller than when processed by another systems. The CARTOMORPH is available for the scientific community, allowing it to be used and improved by researchers and scholars in the field.

KEYWORDS: Digital image processing; mathematical morphology; CARTOMORPH; cartographic features extraction; remote sensing.

LISTA DE ABREVIATURAS E SIGLAS

BTH	Black Top-hat
EE	Elemento Estruturante
MM	Morfologia Matemática
NASA	National Aeronautics and Space Administration
PDI	Processamento Digital de Imagens
RMS	Root Mean Square
SIG	Sistema de Informações Geográficas
WTH	White Top-hat

SUMÁRIO

SUMÁRIO

INTRODUÇÃO

1.1 CONSIDERAÇÕES INICIAIS

Imagens de sensoriamento remoto, provenientes de satélites em órbita terrestre, vêm sendo utilizadas para diversos estudos. Tais estudos só puderam ter início e foram aperfeiçoados seguindo a linha de evolução do processamento digital de imagens (PDI), o qual foi conceitualizado no final da década de 50. Nesta época os Estados Unidos da América, por meio da NASA (*National Aeronautics and Space Administration*), passaram a utilizar os conceitos de PDI em programas computacionais instalados em suas naves espaciais para adquirir e transmitir imagens do espaço para as unidades em solo terrestre. A partir desse momento, diversos satélites foram lançados em órbita terrestre para aquisição de imagens, como um meio alternativo às imagens aéreas. Com o passar dos anos e o aperfeiçoamento das plataformas espaciais, essa alternativa se mostrou interessante, uma vez que o custo de aquisição de imagens aéreas é alto em muitos casos, o que pode tornar inviável a execução de um projeto dependendo da aplicação dele.

Com o lançamento e, consequentemente, a existência, de vários satélites em órbita terrestre, muitos estudos se voltaram para as imagens orbitais adquiridas por estes. Durante tais estudos, percebeu-se a importância do desenvolvimento de rotinas computacionais capazes de extrair, ou seja, obter determinada feição, ou alvo, de interesse presente nas imagens de sensoriamento remoto. O termo feição cartográfica é utilizado, neste contexto, para descrever qualquer alvo ou característica de interesse presente na superfície terrestre e, consequentemente, na imagem adquirida. Dessa forma, a extração de feições presentes em imagens digitais tem sido o propósito de muitos trabalhos científicos na área de processamento digital de imagens (PDI) e visão computacional. Essas pesquisas são motivadas, sobretudo, pela crescente importância dos Sistemas de Informações Geográficas (SIG) e a necessidade de aquisição e atualização de dados espaciais, que são imprescindíveis para o desenvolvimento e manutenção de um SIG (ISHIKAWA; SILVA; NÓBREGA, 2010). Em planejamento urbano existe uma grande necessidade de obtenção de informações atualizadas e acuradas, principalmente, quando as informações são relativas à malha viária, uma vez que órgãos competentes às utilizam para o gerenciamento, planejamento e tomada de decisões (HINZ; BAUMGARTNER, 2000; GALLIS, 2006).

Neste sentido, a automação dos processos de extração de feições cartográficas é de fundamental importância na área de Ciências Cartográficas, todavia, as cenas envolvidas nas imagens adquiridas dificultam este procedimento (DAL POZ, 2005). O advento de imagens de satélite com alta resolução espacial (ex. Ikonos e Quickbird) abriu novas possibilidades para o processo de extração de feições lineares, tais como rodovias (BACHER; MAYER, 2005). Contudo, trabalhar com imagens de satélite é um problema devido à complexidade de sua estrutura e escala. São diversos alvos com diferentes formas, tonalidades e texturas, dos quais cita-se: casas, sombras de edifícios, automóveis

e árvores (PÉTERI; CELLE; RANCHIN, 2003). Desse modo, o processo de extração de feições é um tópico bastante desafiador.

O processo de extração de feições envolve duas tarefas básicas, o reconhecimento e o delineamento. Geralmente, a tarefa de reconhecimento é mais difícil de ser realizada, pois, depende de conhecimentos semânticos para atribuir o significado a cada objeto ou feição presente na imagem. Executado o reconhecimento, cada objeto pode ser delineado geometricamente através de informações geométricas e radiométricas presentes na imagem. Neste contexto, a grande dificuldade é a atribuição de um significado ao objeto de interesse (por exemplo, uma rodovia ou uma drenagem). Essas tarefas são úteis para caracterizar os métodos quanto ao nível de automação. Métodos automáticos desempenham ambas as tarefas: reconhecimento e delineamento, enquanto que em métodos semiautomáticos apenas a tarefa de delineamento é realizada, uma vez que o reconhecimento fica a cargo da habilidade interpretativa do operador (DAL POZ; ZANIN; DO VALE, 2007). No contexto de detecção de feições cartográficas, Vale et al. (2008) citam que nenhuma solução automática se mostrou competitiva frente a habilidade natural do operador humano. Desta forma, soluções semiautomáticas têm sido mais estudadas, como o trabalho de Silva et al. (2012), que combina a habilidade de interpretação humana com a capacidade dos algoritmos computacionais em realizar medidas precisas.

Para a obtenção de bons resultados com a extração de feições cartográficas de interesse é necessário que a estratégia adotada seja eficiente e confiável. Nota-se que vários trabalhos sobre extração de feições empregam métodos lineares para realizar a detecção de bordas, o que não é suficiente para extrair as estruturas geométricas dos objetos presentes em uma imagem e, por isto, o domínio de aplicação desta abordagem torna-se restrito. Desse modo, técnicas não-lineares, como a morfologia matemática (MM), têm sido mais eficientes devido à sua capacidade de remover ruídos e preservar informações de bordas simultaneamente (ISHIKAWA; SILVA; NÓBREGA, 2010). Além disso, processos de extração de feições baseados em MM podem realizar a extração ao analisar as estruturas geométricas dos alvos contidos nas imagens.

Um aspecto importante que se deve atentar quando se deseja realizar processamento morfológico de imagens é o *software* que se utilizará. Com o avanço da tecnologia, os sistemas de processamento e análise de imagens tornam-se obsoletos num período de tempo cada vez mais curto (SILVA; CARRARD; D'ORNELLAS, 2004). Sistemas operacionais e programas, como o Matlab, mudam de versão em média a cada três anos ou menos e, quando isso ocorre, geralmente são necessárias mudanças nas ferramentas utilizadas por estas plataformas. Adicionalmente, a grande maioria dos pacotes, bibliotecas ou *toolkits* para processamento morfológico de imagens que seguem a teoria da MM é de domínio particular, o que impossibilita a modificação dessas ferramentas. Além disso, alguns sistemas existentes possuem limitações quanto ao tamanho da imagem a ser processada. Quando utilizados, esses sistemas forçam o usuário a realizar cortes nas

imagens originais para que estas possam ser processadas. Deste modo, optou-se neste projeto, por desenvolver um sistema computacional (CARTOMORPH) projetado de tal modo que seja capaz de executar as operações sem limitar as dimensões das imagens diretamente, possibilitando que imagens de sensoriamento remoto possam ser processadas sem a necessidade de particionamento e que as tarefas necessárias sejam realizadas em tempo reduzido de processamento. Além disso, a disponibilização do sistema desenvolvido como livre permite que ele seja modificado ou que novas implementações sejam realizadas. Adicionalmente, o sistema tem como ênfase o desenvolvimento de métodos de extração de feições cartográficas utilizando técnicas de processamento morfológico de imagens. O sistema foi desenvolvido em linguagem de programação C/C++ e será disponibilizado com a finalidade de dar suporte à pesquisas e à aplicações na área de cartografia, principalmente no que diz respeito à extração de feições cartográficas de interesse.

1.2 OBJETIVOS

De modo geral, o objetivo deste projeto foi desenvolver um sistema computacional (CARTOMORPH) para processamento digital de imagens, focado no processamento morfológico de imagens, com o intuito de facilitar e melhorar pesquisas de detecção e extração de feições cartográficas, em especial feições do tipo rodovias e pistas de aeroportos.

Os objetivos específicos foram:

- Estudo de técnicas e algoritmos de processamento digital de imagens, principalmente em relação à teoria da morfologia matemática;
- Implementação dos algoritmos estudados em linguagem C/C++ como uma biblioteca de processamento digital de imagens, a qual pode ser utilizada independente, sem necessidade da interface gráfica;
- Implementação de rotina semiautomática para detecção de feições cartográficas de interesse;
- Implementação de metodologia de análise estatística dos resultados obtidos na detecção de feições cartográficas;
- Testes matemáticos de todos os algoritmos implementados no sistema;
- Realização de testes e análises do tempo de execução de cada algoritmo implementado;
- Implementação e testes dos algoritmos sem que estes possuam limitantes relacionados com as dimensões da imagem a ser processada;
- Comparação dos resultados obtidos pelos algoritmos implementados no sistema com resultados de sistemas consolidados no mercado;
- Implementação de uma interface gráfica, trabalhando conjuntamente com a biblioteca de processamento digital de imagens, gerando assim um sistema computacional para o usuário (sistema CARTOMORPH).

1.3 JUSTIFICATIVA

A extração de feições cartográficas de interesse é objeto de estudos em inúmeras pesquisas realizadas pela comunidade cientifica de diversas áreas. Atualmente, há diferentes tipos de métodos propostos, os quais diferem em suas propriedades matemáticas e algorítmicas. Porém, os resultados apresentados estão longe de serem satisfatórios (BELLENS et al., 2008). No processo de extração de feições, constata-se que a tendência é a constante diminuição da dependência do operador humano, exigindo cada vez mais metodologias automáticas (DAL POZ; ZANIN; DO VALE, 2007).

A utilização da teoria de Morfologia Matemática como método alternativo para extração de feições vem se mostrando bastante eficaz. A MM se mostrou uma excelente ferramenta de extração de informação a partir da análise das estruturas geométricas dos alvos, conforme apresentam os trabalhos publicados no meio científico por pesquisadores espalhados pelo mundo (YAN; ZHAO, 2003; GÉRAUD; MOURET, 2004; MOHAMMADZADEH; TAVAKOLI; ZOEJ, 2006; BELLENS et al., 2008) e pelo grupo de MM da Faculdade de Ciências e Tecnologia (FCT/UNESP - Presidente Prudente) (STATELLA; SILVA, 2008; ISHIKAWA; SILVA; NÓBREGA, 2010; RODRIGUES; SILVA; LEONARDI, 2010; SANTOS; SILVA; NÓBREGA, 2010), os quais confirmam a utilização da MM como base de métodos de extração de feições.

Atualmente, a *Toolbox* comercial de Morfologia Matemática disponível para o processamento morfológico de imagens, desenvolvida pela *SDC Information Systems*, é executada acoplada ao *software* Matlab, sendo ambos, soluções de *software* comerciais e de acesso restrito aos usuários. Uma das grandes limitações desta *Toolbox* é que os usuários não possuem acesso as equações morfológicas, o que dificulta substancialmente pesquisas aprofundadas na área que necessitem acesso e até mesmo realizar modificações ou adaptações em determinados métodos. Assim, o desenvolvimento de um sistema computacional aberto de processamento digital de imagens, focado no processamento morfológico (CARTOMORPH), justifica a relevância do trabalho. Tal fato se confirma uma vez que ele disponibiliza para a comunidade científica acesso ao código fonte, o que permite a alteração das operações morfológicas, bem como a proposição de novos algoritmos e funcionalidades. Para facilitar neste aspecto, foi produzida, em conjunto com o sistema CARTOMORPH, uma documentação sobre todos os operadores implementados e características do sistema em desenvolvimento.

FUNDAMENTAÇÃO TEÓRICA

2.1 IMAGENS DIGITAIS

Uma imagem digital é utilizada para estabelecer uma ligação entre o usuário e os processos da computação gráfica. Pode-se afirmar ainda que a imagem digital está presente em todos os processos da computação gráfica, seja como parte do processo, ou como resultado final dele (GOMES; VELHO, 1994).

O termo imagem pode ser definido como um arranjo de elementos sob a forma de uma matriz, onde cada célula dessa matriz possui sua posição definida de acordo com um sistema de coordenadas possuindo linhas e colunas representadas, respectivamente, por “x” e “y”. Cada uma dessas células é chamada de *pixel* (do inglês *picture element*) e armazena um valor, representado por “z”, que corresponde ao valor de brilho, ou o nível de cinza associado aquela posição da imagem (CROSTA, 1999). O conceito de imagem digital pode ser resumido ao dizer que cada *pixel* da mesma possui uma coordenada (x, y) onde armazena um valor de brilho, intensidade ou cor da imagem em questão (BAXES, 1994).

A área de estudo da cartografia utiliza muitas imagens digitais obtidas por câmaras aéreas, sensores aerotransportados ou sensores orbitais. Um dos grandes avanços nas imagens de sensores orbitais é o aumento da resolução espacial da imagem, que hoje pode ser menor do que um metro em alguns sensores (NÓBREGA, 2007).

Fotografias aéreas possuem, geralmente, resolução espacial melhor do que imagens orbitais, porém, o custo de aquisição dessas imagens é maior que o custo de aquisição de imagens orbitais, sendo assim, as imagens orbitais estão levando vantagem em relação às fotografias aéreas (ISHIKAWA, 2008).

O volume de informações contido em uma imagem é muito grande e, por isso, surgiram vários formatos de armazenamento de imagens, sendo que cada um deles pode possuir um método de compressão de dados próprio ou até mesmo não possuir compressão (LILLESAND; KIEFER; CHIPMAN, 2007). Contudo, esses métodos de compressão, ao serem utilizados podem ocasionar perda das informações presentes na imagem. Portanto, os testes realizados durante esse projeto foram feitos com utilização de imagens no formato *bitmap* (*.bmp*), um formato que não possui compressão de dados, fazendo com que melhores resultados na análise pudessem ser obtidos.

2.2 PROCESSAMENTO DIGITAL DE IMAGENS

As técnicas de Processamento Digital de Imagens (PDI) envolvem a manipulação e interpretação das informações contidas na imagem com o auxílio de um computador (LILLESAND; KIEFER; CHIPMAN, 2007). O objetivo do PDI é fazer com que as imagens melhorem sua qualidade visual (BANON, 1987). No entanto, outros autores afirmam que o objetivo principal dessas técnicas pode ser dividido em aprimorar a qualidade e/ou restaurar informações contidas em uma imagem digital (GONZALEZ; WOODS, 2010). Tal fato pode ocorrer por várias técnicas de PDI, assim como modificações de contraste, filtragem de ruídos, correções de distorções, entre outras. As técnicas de PDI, implementadas nesse projeto, serão descritas nos próximos tópicos.

2.2.1 Binarização

O operador da binarização consiste basicamente em converter uma imagem em tons de cinza, a qual tem seu nível digital variando de 0 a 2^n -1, para uma imagem binária, a qual possui apenas dois tons, branco e preto, como níveis digitais aceitáveis. Para realizar tal operação primeiramente deve-se escolher um valor de limiar *T*, com o qual o operador transformará todo pixel para branco, no caso do valor digital ser maior ou igual a *T*, ou para preto, no caso de ser menor que *T* (MCANDREW, 2004). Esta operação está apresentada na Equação (1).

$$f'(x,y) = \begin{cases} 2^n - 1, & se\, f(x,y) >= T \\ 0, & se\, f(x,y) < T \end{cases} \quad (1)$$

2.2.2 Inversão

A operação de inversão consiste em inverter os valores digitais da imagem. Em uma imagem binária, os valores que são pretos são transformados para brancos e vice-versa. Já em imagens variando em tons de cinza, a operação consiste em subtrair o valor digital atual da imagem em relação ao valor máximo permitido para aquela imagem (2^n-1), como apresentado na Equação (2). Inverter os níveis de intensidade de uma imagem, como proposto por esta técnica, cria o equivalente a um negativo fotográfico (GONZALEZ; WOODS, 2010).

$$f'(x, y) = (2^n - 1) - f(x, y) \quad (2)$$

2.2.3 Filtros de Suavização ou Borramento

Os filtros espaciais de suavização são baseados em operações realizadas na vizinhança de um determinado *pixel*. O termo filtro deve ser observado como uma função que realiza determinada operação com os valores digitais de intensidade dos *pixels* vizinhos a um determinado *pixel* central, o qual terá seu valor alterado para o valor resultante da operação realizada (GONZALEZ; WOODS; EDDINS, 2004).

De maneira breve, um filtro de suavização espacial consiste em analisar uma vizinhança, a qual normalmente é definida como um pequeno retângulo, e uma operação, determinada previamente, realizada sobre os *pixels* incluídos nesta vizinhança. Dessa forma, o filtro define o *pixel* resultante com coordenadas do centro da vizinhança definida e valor intensidade com o resultado da operação de filtragem (GONZALEZ; WOODS, 2010).

Dentre os filtros de suavização existentes na literatura, os filtros da Média e da Mediana são de fácil entendimento e de grande uso em aplicações reais. A filtragem da média consiste em substituir cada *pixel* da imagem pela média do nível de intensidade de seus vizinhos definidos previamente. Já a filtragem da mediana substitui o valor do *pixel* central da máscara pelo valor mediano dentre todos os valores dos seus vizinhos. Deve-se notar que a aplicação de cada filtro de suavização produz uma nova imagem a partir da imagem original, sendo assim, o valor de cada *pixel* da imagem resultante depende apenas dos *pixels* da imagem original. Em outras palavras, o resultado obtido para determinado *pixel* pela média, mediana, ou qualquer outra operação, não afeta os resultados que serão obtidos para os *pixels* restantes na operação, até que a mesma seja terminada (PEDRINI; SCHWARTZ, 2008).

2.2.4 Filtro Laplaciano

A detecção de bordas é um dos temas mais comuns para a análise de imagens digitais, por esse fato, este tema provavelmente possua mais algoritmos propostos na literatura do que qualquer outro assunto particular (PARKER, 2010). Essa afirmação não é atual e pode não mostrar a real situação da literatura. No entanto, a partir dessa afirmação é possível identificar a importância deste assunto na literatura.

As bordas de um alvo presente em uma imagem podem ser consideradas como uma variação local da intensidade do nível de cinza. Dessa forma, o gradiente $\nabla f\,(x,\ y)$ apresentado na Equação (3), demonstra essa variação local de intensidade.

$$\nabla f\left(x,y\right)=\left[\frac{\partial f}{\partial x}\,\frac{\partial f}{\partial y}\right]^{T}=\left[f_x f_y\right]^{T} \tag{3}$$

O filtro Laplaciano, por ser uma derivada de segunda ordem, é muito sensível a ruídos, detectando assim qualquer leve mudança nos níveis de cinza de intensidade (PEDRINI; SCHWARTZ, 2008). Computacionalmente, o gradiente e, consequentemente, o

filtro Laplaciano, pode ser calculado a partir da convolução de determinada máscara sobre a imagem original. Esta operação de convolução e a máscara utilizada para este filtro estão representadas na Equação (4).

$$\mathrm{Im}_{Laplace} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} * \mathrm{Im}_{Entrada} \quad (4)$$

2.3 MORFOLOGIA MATEMÁTICA

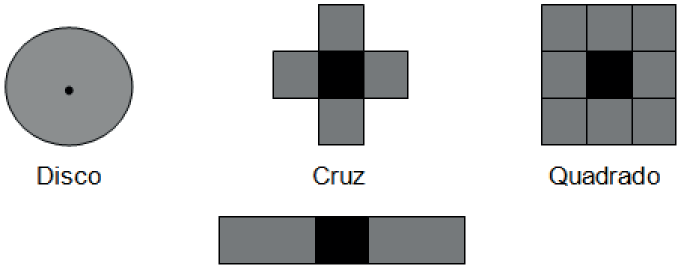
A Morfologia Matemática (MM) surgiu por volta de 1964 com trabalhos de Matheron e Serra na Escola Superior de Minas de Paris localizada em Fontainebleau (BANON; BARRERA, 1998). A palavra morfologia é composta pelas palavras gregas *morphos* (formas) e *logia* (estudo), ou seja, a morfologia baseia-se no estudo da forma que a matéria assume (FACON, 1996).

De acordo com Soille (2003), a MM pode ser definida como uma teoria para análise das estruturas espaciais. Ela é chamada de morfologia, pois consiste na análise da forma e da estrutura dos objetos. Ela é matemática no sentido que a análise se baseia na teoria de conjuntos, integrais geométricas e álgebra booleana. A MM não é somente uma teoria, mas também uma poderosa ferramenta de análise de imagens, em particular para aquelas aplicações em que aspectos geométricos são relevantes. A ideia principal da MM é analisar a forma dos objetos através de um modelo geométrico denominado elemento estruturante (GOUTSIAS; HEIJMANS, 2000).

2.3.1 Elemento Estruturante

O elemento estruturante (EE) pode ser definido como um conjunto completamente definido e conhecido (forma e tamanho), o qual é comparado, a partir de uma transformação, ao conjunto desconhecido da imagem (FACON, 1996). O resultado desta transformação permite avaliar o conjunto desconhecido. Este elemento é a chave para o sucesso das operações, desde que seja escolhido de forma adequada. Para selecionar o elemento estruturante mais adequado, pode-se considerar a forma dos objetos, ou definir um tamanho específico e, para alguns elementos, considera-se uma orientação específica (SOILLE, 2003). Determinar o tamanho e a forma de um elemento estruturante é um processo predominantemente empírico. Porém, a seleção de um elemento estruturante depende das formas geométricas do objeto a ser extraído na imagem. A Figura 1 apresenta quatro tipos de elementos estruturantes possíveis. Cada elemento estruturante possui um ponto origem, representado na imagem pela cor mais escura, o qual é responsável pelo posicionamento do elemento estruturante sobre determinado *pixel* durante a execução de qualquer operação morfológica. Neste trabalho, todos estes tipos de elementos estruturantes apresentados foram implementados e serão apresentados posteriormente.

Figura 1 - Exemplos de elementos estruturantes.



Fonte: *Soille (2003)*.

2.3.2 Operadores Elementares

A Morfologia Matemática é constituída a partir de dois operadores básicos denominados erosão e dilatação (MATHERON, 1974; SERRA; CRESSIE, 1982). Estes operadores são a base para a construção de outros operadores morfológicos, como a abertura, fechamento, gradiente morfológico, entre outros.

As operações morfológicas foram elaboradas inicialmente para imagens binárias, sendo posteriormente ampliadas para imagens em tons de cinza e atualmente para imagens coloridas. Neste trabalho, as operações morfológicas foram implementadas para imagens binárias e, quando possível, para imagens em tons de cinza e coloridas em modo RGB. Devido à diferença de enfoque entre os dois tipos de imagens, torna-se necessário distinguir essas operações.

Na MM binária, as imagens são representadas por conjuntos pertencentes a Z^2, cujas coordenadas apresentam características (x, y) e as operações baseiam-se na teoria de conjuntos. Na morfologia cinzenta, ou seja, aquela realizada sobre imagens em tons de cinza, as imagens são tratadas como funções, pertencem a Z^3 e as coordenadas passam a assumir a forma (x, y, z), onde z corresponde à intensidade, e os operadores são baseados na teoria de reticulados (operações então assumem valores máximo e mínimo) (FACON, 1996). Na sequência, serão discutidas as transformações morfológicas elementares, denominadas erosão e dilatação, sobre imagens binárias e em níveis de cinza, os filtros morfológicos de abertura e fechamento, e outros operadores criados com a técnica da morfologia matemática.

2.3.3 Dilatação e Erosão Binárias

A dilatação binária de um conjunto X pelo elemento estruturante B é denotada por $\delta_B(X)$ e é definida como a posição de todos os pontos de B, quando sua origem é posicionada sobre todos os pontos do alvo de interesse, presente em X (SOILLE, 2003), como apresentado na Equação (5).

$$\delta_B(X) = \{x \mid B_x \cap X \neq \varnothing\} \tag{5}$$

Por esta definição, o elemento estruturante *B* percorre a imagem verificando para cada ponto uma possível interseção da vizinhança com o alvo de interesse presente em *X*. Caso seja verdadeiro, as coordenadas do ponto origem de *B*, na imagem resultante, será um *pixel* relevante (1), caso contrário será irrelevante (0). Os efeitos da dilatação binária são: aumento de partículas; preenchimento de pequenos buracos e conexão de grãos próximos (FACON, 1996).

Já a erosão binária de um conjunto *X* por um elemento estruturante *B* é denotada por $\varepsilon_B(X)$ e é definida como as posições dos pontos origem, *x*, tal que *B* está contido no alvo de interesse presente em *X* (SOILLE, 2003), como apresentado pela Equação (6).

$$\varepsilon_B(X) = \{x \mid B_x \subseteq X\} \tag{6}$$

O elemento estruturante *B* desliza sobre a imagem *X*, comparando cada pixel com a vizinhança de *x*. Se todos os *pixels* de *B* corresponderem ao alvo de interesse em *x* preserva-se o *pixel* central como parte do alvo de interesse. Em geral, a erosão binária apresenta os seguintes efeitos: diminuição de partículas; eliminação de grãos de tamanho inferior ao tamanho do elemento estruturante; aumento dos buracos e permite a separação de grãos próximos (FACON, 1996).

2.3.4 Dilatação e Erosão Dilatação em Níveis de Cinza

A dilatação em níveis de cinza da imagem *f* pelo elemento estruturante *B* é denotada por $\delta_B(f)$ e definida como o máximo valor da imagem na janela definida pelo elemento estruturante, quando sua origem está em *x* (SOILLE, 2003), como apresentado pela Equação (7).

$$[\delta_B(f)](x) = \max_{b \in B}\{f(x+b) + B(b)\} \tag{7}$$

Os efeitos visuais da dilatação em níveis de cinza são: clareamento da imagem; alargamento e aumento dos padrões claros; conexão dos padrões claros próximos; redução ou eliminação dos padrões escuros e separação dos padrões escuros próximos (FACON, 1996).

A erosão em níveis de cinza de uma imagem *f* por um elemento estruturante *B* é denotada por $\varepsilon_B(f)$ e é definida da seguinte maneira: o valor da erosão num dado *pixel x* é o valor mínimo da imagem na janela definida pelo elemento estruturante, quando sua origem está em *x* (SOILLE, 2003), como apresentado pela Equação (8).

$$[\varepsilon_B(f)](x) = \min_{b \in B}\{f(x+b) - B(b)\} \tag{8}$$

Os efeitos visuais da erosão em níveis de cinza são: escurecimento da imagem; alargamento e aumento dos padrões escuros; conexão dos padrões escuros próximos; redução ou eliminação dos padrões claros e separação dos padrões claros próximos (FACON, 1996).

2.3.5 Filtros Morfológicos de Abertura e Fechamento

A abertura de uma imagem *f* por um elemento estruturante *B* é denotada por $\gamma_B(f)$, sendo definida como a erosão de *f* por *B*, ($\varepsilon_B(f)$) seguida da dilatação com o elemento estruturante transposto $\breve{B}$ ($\delta_{\breve{B}}$) (SOILLE, 2003), como definido na Equação (9).

$$\gamma_B(f) = \delta_{\breve{B}}[\varepsilon_B(f)] \tag{9}$$

Os efeitos da abertura binária são: a abertura não devolve, de forma geral, o conjunto inicial; separa as partículas; elimina partículas com tamanho inferior ao elemento estruturante; as entidades restantes após abertura ficam quase idênticas às originais e o conjunto aberto é menos rico em detalhes que o conjunto inicial. A definição de abertura binária aplica-se também sobre imagens em níveis de cinza (FACON, 1996).

O fechamento da imagem *f* pelo elemento estruturante *B* é denotado por $\phi_B(f)$ e é definido como a dilatação de *f* com o elemento estruturante *B* ($\delta_B(f)$) seguida da erosão com o elemento estruturante transposto $\breve{B}$ ($\varepsilon_{\breve{B}}$) (SOILLE, 2003), como definido na Equação (10).

$$\varphi_B(f) = \varepsilon_{\breve{B}}[\delta_B(f)] \tag{10}$$

Os efeitos do fechamento binário são: suavização das fronteiras pelo exterior; preenchimento dos buracos no interior das partículas com tamanho inferior ao elemento estruturante; emenda de partículas próximas; as entidades restantes após fechamento ficam quase idênticas às da imagem original e o conjunto fechado é menos rico em detalhes que o conjunto inicial. Como no processo de abertura, toda formulação do fechamento para imagens binárias é aplicada para imagens em tons de cinza (FACON, 1996).

2.3.6 Operador de Top-hat

Pode-se explorar as propriedades das aberturas e dos fechamentos, apresentadas anteriormente, para construir novos operadores morfológicos. Como a abertura e o fechamento removem estruturas da imagem que não contenham o elemento estruturante, essas podem ser recuperadas a partir da subtração da imagem original e sua abertura ou entre o fechamento e a imagem original. Essas operações são conhecidas como *top-hat's* (STATELLA, 2012).

São definidos dois tipos básicos de operadores *top-hat*. O *top-hat* por abertura (*white top-hat* – WTH) e o *top-hat* por fechamento (*black top-hat* – BTH). O *top-hat* por abertura de uma imagem pode ser definido como a diferença entre a imagem original e sua abertura, como pode ser visto na Equação (11).

$$WTH(f) = f - \gamma(f) \qquad (11)$$

Uma vez que a abertura é um operador anti-extensivo, note que os valores do WTH serão sempre maiores ou iguais a zero (0) (SOILLE, 2003).

Por outro lado, o *top-hat* por fechamento pode ser definido como sendo a diferença entre o fechamento da imagem original por ela mesma, como definido na Equação (12).

$$BTH(f) = \phi(f) - f \qquad (12)$$

Em oposição à abertura, o operador de fechamento é extensivo, garantindo então que os valores do BTH serão sempre maiores ou iguais a zero (0) (SOILLE, 2003). Note que os operadores de *top-hat* por abertura e fechamento são complementares, como apresentado na Equação (13).

$$BTH = WTH^{C} \qquad (13)$$

2.3.7 Gradientes Morfológicos

Os gradientes morfológicos são conhecidos e utilizados como detectores de bordas. Em imagens binárias, o princípio básico do gradiente consiste em representar as variações de branco para preto, ou preto para branco, que ocorrem na imagem. Vários gradientes foram desenvolvidos utilizando morfologia matemática, sendo o mais comum deles conhecido simplesmente como gradiente morfológico ou gradiente morfológico total (DOUGHERTY; LOTUFO, 2003). Para obter o resultado deste gradiente é necessário subtrair o resultado da erosão de uma imagem do resultado da dilatação da mesma imagem, como definido na Equação (14).

$$grad(f) = \delta_B - \varepsilon_B \qquad (14)$$

Onde *B* é um elemento estruturante com centro na origem. Esse gradiente pode ser definido como a soma de dois outros gradientes, o gradiente interno (gradiente de erosão) e o gradiente externo (gradiente de dilatação), os quais estão definidos, respectivamente, nas Equações (15) e (16).

$$G_e = grad_{\text{int}} = f - \varepsilon_B \quad (15)$$
$$G_d = grad_{\text{ext}} = \delta_B - f \quad (16)$$

Nota-se que os gradientes definidos anteriormente dependem da dimensão e forma do elemento estruturante *B* escolhido. Além disso, as mesmas equações podem ser generalizadas para o uso de imagens em tons de cinza (DOUGHERTY; LOTUFO, 2003).

Além do gradiente morfológico citado, outros detectores de bordas utilizando morfologia matemática foram propostos e desenvolvidos. Entre eles, podem-se citar os operadores G_{min} (Combinado Mínimo), G_{max} (Combinado Máximo), G_{sum} (Combinado da Soma) e G_{blur} (Combinado de Borramento Mínimo), os quais estão definidos respectivamente pelas Equações (17), (18), (19) e (20) (LEE; HARALICK; SHAPIRO, 1987; MASCARENHAS; SILVA, 1990).

$$G_{\text{min}} = \min\,(G_e\,,\,G_d) \quad (17)$$
$$G_{\text{max}} = \max\,(G_e,\,G_d) \quad (18)$$
$$G_{sum} = G_e + G_d \quad (19)$$
$$G_{blur} = \min\,(I - \varepsilon\,(I),\,\delta\,(I) - I) \quad (20)$$

Onde, *I* é obtida por um filtro de suavização da imagem de entrada por um elemento estruturante caixa, ou seja, um elemento estruturante quadrado de dimensões 3x3, como apresentado pela Figura 1.

2.4 EXTRAÇÃO DE FEIÇÕES CARTOGRÁFICAS

As chamadas feições cartográficas são definidas como qualquer alvo, objeto ou característica de interesse, presente em uma imagem digital. Como exemplo, podem-se citar rodovias, corpos d'água, trechos urbanos e crateras de impacto em imagens planetárias.

O estudo de extração de feições cartográficas tem por objetivo processar a imagem digital por meio de técnicas de PDI, para identificar e reconhecer os alvos de interesse presentes na imagem. Sendo assim, o processo de extração de feições cartográficas pode ser considerado como uma técnica de reconhecimento de padrões. Este nome é dado às técnicas e aos algoritmos desenvolvidos com o objetivo de identificar alvos em imagens digitais. Essas técnicas são consideradas como a ciência da arte de nomear objetos naturais do mundo real analisando uma imagem digital (LAMPINEN; LAAKSONEN; OJA, 1998).

O termo extração pode ter variados significados nas diferentes comunidades que trabalham com análise de imagem. No presente contexto, duas tarefas distintas estão relacionadas ao termo extração, as tarefas de reconhecimento e delineamento. A tarefa de reconhecimento procura imitar a habilidade natural do homem para identificar o alvo, ou feição, de interesse presente na imagem. Já a segunda tarefa, a de delineamento, delimita

a área de cada alvo presente na imagem por meio do contorno do alvo identificado. Métodos que visam a extração automática da feição de interesse devem integrar ambas as tarefas, enquanto que métodos semiautomáticos realizam somente a tarefa de delineamento, deixando ao operador humano a tarefa interpretativa de reconhecimento do alvo (DAL POZ; DO VALE; ZANIN, 2005).

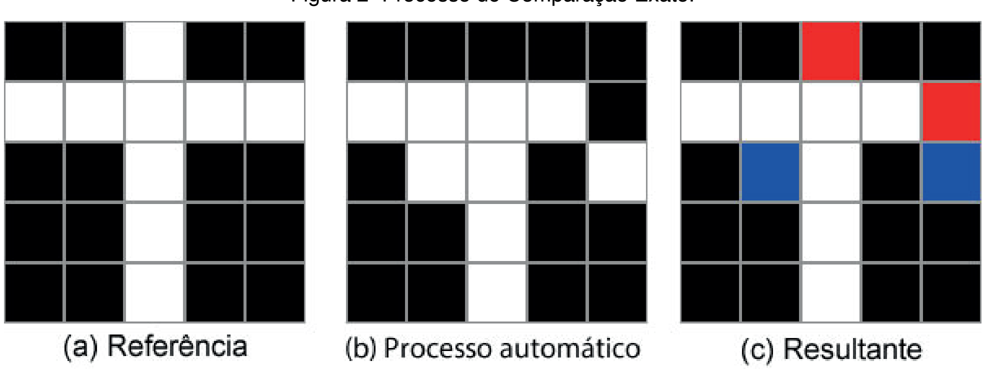
2.4.1 Análise Estatística de Extrações Cartográficas

Tendo obtido o resultado da extração, a avaliação do resultado obtido é de extrema importância. Na literatura é possível encontrar métricas destinadas à avaliação estatística de feições cartográficas lineares como por exemplo a para malha viária (WIEDEMANN et al., 1998; WIEDEMANN, 2003), ou adaptações dessas métricas implementadas em alguns algoritmos (CARDIM; SILVA, 2011, 2013; SILVA; CARDIM, 2012).

Nas avaliações de processos de extração encontradas é sempre necessária a utilização de uma imagem de referência, a qual é utilizada como base dos cálculos estatísticos, sendo sempre considerada como correta. Durante a avaliação, a imagem resultante do processo de extração é comparada *pixel* a *pixel* com a imagem de referência. Essa comparação pode ocorrer tanto de maneira precisa (CARDIM; SILVA, 2011; SILVA; CARDIM, 2012), como com a utilização de um *buffer*, ou área, de tolerância (WIEDEMANN, 2003; CARDIM; SILVA, 2013).

A comparação de modo preciso, ou exato, é exemplificada pela Figura 2. Como explicado anteriormente, para aplicar essas comparações necessita-se de uma imagem de referência, Figura 2 (a), a qual será comparada com o resultado da extração automática, Figura 2 (b), gerando a uma imagem resultante, Figura 2 (c). Na imagem resultante da comparação exata, apresentada pela Figura 2, os pontos vermelhos representam os falsos negativos e os pontos azuis os falsos positivos.

Figura 2- Processo de Comparação Exato.



Fonte: Cardim e Silva (2011)

Com base no processo de comparação exato apresentado, foi estabelecida a métrica de correspondência (C) entre as imagens apresentada na Equação (21).

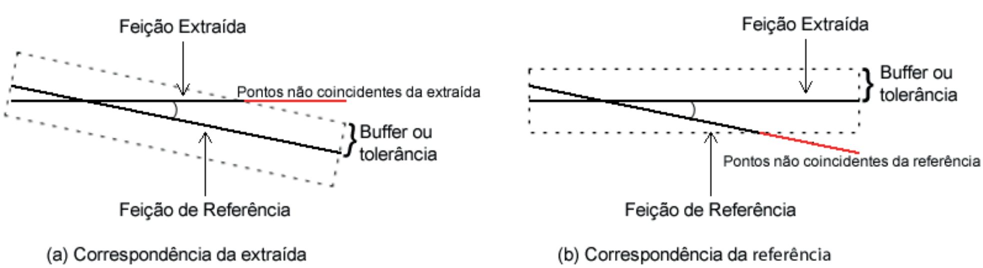
$$C = \frac{TB}{TB + TA + TV} \quad (21)$$

Onde,

- C: Valor da correspondência;
- TB: Total de *pixels* coincidentes;
- TA: Total de falsos positivos (pontos azuis);
- TV: Total de falsos negativos (pontos vermelhos).

Por outro lado, a comparação utilizando-se da área de tolerância é exemplificada pela Figura 3. Neste processo, primeiramente é criado uma área de tolerância, ou *buffer*, ao redor do alvo de interesse presente na imagem de referência e posteriormente realiza-se a comparação da imagem resultante do processo automático com a imagem de referência, como apresentado na Figura 3 (a). Do mesmo modo, um *buffer* é gerado ao redor do alvo de interesse presente na imagem resultante do processo automático e essa imagem é comparada com a imagem de referência, Figura 3 (b). Na Figura 3, os pontos em vermelho em ambas as comparações são considerados como erros obtidos pelo processo de extração, sendo em (a) os falsos positivos e em (b) os falsos negativos.

Figura 3 - Processo de Comparação com Área de Tolerância.



Fonte: Adaptado de Wiedemann (2003)

Com base no processo de comparação com área de tolerância, é possível realizar cálculos estatísticos que permitam a avaliação do resultado obtido pelo processo de extração. Tais métricas serão apresentadas nos próximos tópicos.

2.4.1.1 Completeness

Essa métrica, apresentada na Equação (22), representa a porcentagem de *pixels* da imagem de referência que foram extraídos corretamente pelo método de extração. O valor pode variar no intervalo [0:1], sendo 1 o valor ótimo a ser obtido (WIEDEMANN, 2003).

$$completeness = \frac{\text{total coincidentes da referência}}{\text{total da feição de referência}} \quad (22)$$

2.4.1.2 Correctness

Apresentada na Equação (23), essa métrica representa a porcentagem de *pixels* da imagem extraída que coincidem com a imagem de referência. O valor resultado pode variar no intervalo [0:1], tendo o valor 1 como ótimo (WIEDEMANN, 2003).

$$correctness = \frac{\text{total coincidentes da extração}}{\text{total da feição de extração}} \quad (23)$$

2.4.1.3 Quality

Definida na Equação , a métrica *Quality* é obtida a partir dos resultados das equações de *Completeness* e *Correctness* e portanto não incorpora mais informações do que as informações presentes nessas duas métricas. No entanto, a métrica *Quality* pode ser útil quando for necessário fazer uma análise com apenas um valor. Assim como as métricas anteriores, o valor resultante pode variar no intervalo [0:1], tendo 1 como resultado ótimo (WIEDEMANN, 2003).

$$quality = \frac{completeness * correctness}{\text{completeness - completeness * correctness + correctness}} \quad (24)$$

2.4.1.4 Redundancy

Definida pela Equação (25), esta métrica representa a porcentagem de *pixels* que são redundantes em relação a análise, ou seja, aqueles que possam ter sido considerados como erros em ambas as métricas *Completeness* e *Correctness*. O valor ótimo para essa métrica é 0, tendo como intervalo possível [-∞:1] (WIEDEMANN, 2003).

$$redundancy = \frac{\text{coincidentes da extração - coincidentes da referência}}{\text{total da feição extraída}} \quad (25)$$

2.4.1.5 RMS

O RMS (*Root Mean Square*) compreende a diferença média entre os *pixels* coincidentes da extração e os da referência. Por depender da dimensão da área de tolerância, essa métrica normalmente possui uma distribuição normal da feição extraída sobre a área de tolerância ao redor da feição de referência. Dessa forma, o cálculo pode ser simplificado para a Equação (26), a qual possui como intervalo de variação [0:dimensão do *buffer*] e valor ótimo igual a 0 (WIEDEMANN, 2003).

$$RMS = \frac{1}{\sqrt{3}} * (\text{Dimensão do buffer}) \quad (26)$$

MATERIAIS E MÉTODOS

3.1 MATERIAIS

Os materiais utilizados neste projeto foram:

- Microcomputador do tipo PC;
- Compiladores para as linguagens C e C++ (gcc e g++);
- Ambiente de desenvolvimento Netbeans e C++ Builder;
- Biblioteca EasyBMP para trabalhar com imagens do tipo .bmp.
- Imagens de sensoriamento remoto da base de dados de imagens da FCT/UNESP - Presidente Prudente;
- *Toolbox* de Morfologia Matemática desenvolvida pela SDC *Information System;*
- Sistemas computacionais adicionais como o Matlab.

3.2 MÉTODO PARA IMPLEMENTAÇÃO DO CARTOMORPH

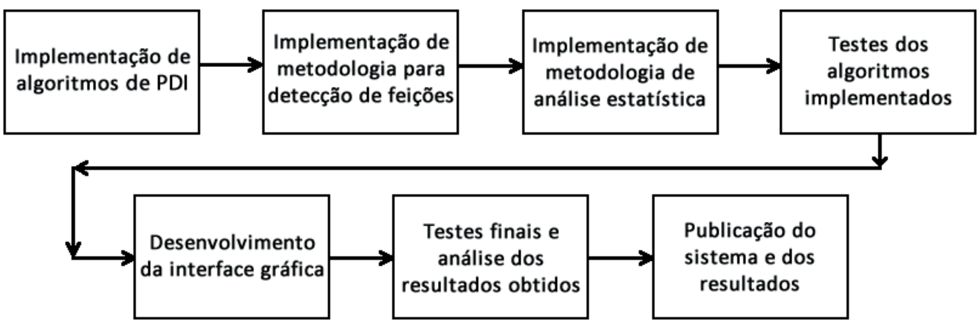
A metodologia empregada no projeto de desenvolvimento do sistema computacional (CARTOMORPH) fundamenta-se em modelos matemáticos e algoritmos de processamento digital de imagens para viabilizar a implementação de rotinas de extração de feições cartográficas; sobretudo rodovias e pistas de aeroportos.

Para atingir os objetivos propostos nesta pesquisa, algumas etapas foram realizadas, sendo estas listadas a seguir e simplificadas pelo fluxograma apresentado pela Figura 4.

- Implementação de algoritmos gerais de processamento digital de imagens necessários no sistema em linguagem de programação C;
- Implementação de metodologia semiautomática para detecção de feições cartográficas de interesse;
- Implementação de metodologia de análise estatística dos resultados obtidos na detecção de feições cartográficas;
- Testes dos algoritmos de PDI implementados, verificando a sua eficácia matematicamente;
- Junção de todos os algoritmos implementados por meio da linguagem de programação C++, criando-se uma biblioteca de funções e ferramentas para processar imagens sem o uso de interface gráfica;
- Testes com todos as funcionalidades disponíveis na biblioteca de funções;
- Comparação dos resultados obtidos pela biblioteca desenvolvida com os resultados de outros sistemas computacionais que possuem as mesmas funções testadas (como o MATLAB);

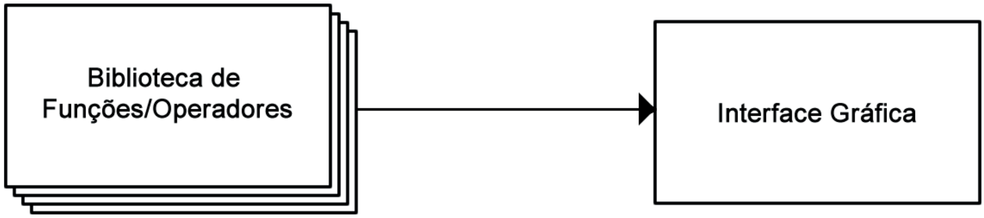
- Desenvolvimento de uma interface gráfica, fazendo uso do C++ Builder, que possibilita a manipulação de todas as funções da biblioteca implementada de modo simples e intuitivo;
- Testes finais da interação entre a interface desenvolvida e as funções da biblioteca criada;
- Análise dos resultados obtidos com a implementação do CARTOMORPH;
- Elaboração de documentação auxiliar, na qual todos os operadores implementados no sistema foram definidos e exemplificados;
- Disponibilização da biblioteca de funções, do sistema CARTOMORPH e da sua documentação para a comunidade científica, possibilitando a implementação de novas funcionalidades desejadas dentro do sistema.

Figura 4 – Fluxograma da metodologia de desenvolvimento.



O desenvolvimento do sistema foi realizado de tal modo que a biblioteca de funções possa ser utilizada independentemente da interface gráfica. Dessa forma, o sistema não fica dependente de uma única interface e um único ambiente de desenvolvimento, uma vez que a biblioteca de funções e operadores pode ser utilizada independentemente da interface, a qual apenas utiliza os operadores da biblioteca implementada. Essa estrutura pode ser visualizada na Figura 5. A ideia é fazer com que usuários familiarizados com programação de computadores possam utilizar diretamente a biblioteca de funções de modo a facilitar o desenvolvimento de novas funcionalidades e adaptações ou melhorias das funções já existentes. Por outro lado, os usuários que não possuem conhecimento de programação de computadores podem utilizar a interface gráfica desenvolvida para aplicar todos os operadores presentes no sistema CARTOMORPH.

Figura 5 – Estrutura de implementação.



Como um dos objetivos do projeto é possibilitar o acesso livre ao sistema desenvolvido, é necessário a existência de uma documentação mostrando os detalhes da implementação dos algoritmos e do modo de uso pelo usuário. Esta documentação está disponível juntamente com o sistema CARTOMORPH possibilitando que usuários utilizem o sistema de forma correta e até desenvolvam melhorias relacionadas diretamente com a implementação dos algoritmos, uma vez que todo o código também está disponível.

Tendo em vista o tempo para desenvolvimento do projeto, foram definidas as funcionalidades iniciais do sistema, ou seja, os algoritmos a serem implementados no sistema CARTOMORPH, os quais estão listados a seguir.

- Binarização;
- Inversão;
- Erosão morfológica;
- Dilatação morfológica;
- Abertura e abertura por área;
- Fechamento e fechamento por área;
- Filtros GMin, GMax, GSum e GBlur;
- Filtros de suavização (média, mediana e bilateral);
- Detectores de bordas (Filtro laplaciano e gradientes morfológicos);
- Esqueletonização ou afinamento;
- Análise estatística de métodos de extração;
- Equalização de histograma;
- TopHat por fechamento e abertura.

Contudo, ao longo do desenvolvimento do projeto, percebeu-se a necessidade da existência de funcionalidades que não foram previstas inicialmente no projeto, mas devido à sua importância foram desenvolvidas. Tais funcionalidades estão listadas a seguir.

- Rotulação de objetos presentes na imagem;
- Dilatação e erosão condicional;
- Filtros de suavização (Gaussiano);
- Operador de Crescimento por região;
- Metodologia semiautomática para detecção de feições cartográficas.

Vale ressaltar que as metodologias para detecção de feições e para análise estatística foram implementadas não só como funcionalidades extras ao usuário, mas também para comprovar a importância e a utilização do sistema desenvolvido na Cartografia. Além disso, a implementação dessas funções demonstra que pesquisas específicas para o desenvolvimento de metodologias automáticas para extração de feições cartográficas são beneficiadas com a utilização do CARTOMORPH. Os dois conjuntos de funcionalidades listadas anteriormente representam as principais funcionalidades do sistema desenvolvido. No entanto existem outras funcionalidades secundárias que foram desenvolvidas e implementadas de acordo com a necessidade.

3.3 MÉTODO PARA VALIDAÇÃO DO CARTOMORPH

A análise da correta implementação do sistema computacional CARTOMORPH foi realizada matematicamente em cada funcionalidade de PDI desenvolvida para garantir o correto funcionamento e validar o sistema desenvolvido. No caso específico do algoritmo de detecção de feições, a análise dos resultados da detecção foi analisada estatisticamente pelos métodos apresentados no item 2.4.1. da fundamentação teórica.

Análises referentes à complexidade de cada algoritmo foram realizadas com o intuito de minimizar o custo computacional, tanto em processamento quanto em uso de memória, o que minimizou o tempo de execução de cada operação.

APRESENTAÇÃO DOS RESULTADOS

Nesta seção serão apresentados os resultados obtidos pelos algoritmos implementados no sistema desenvolvido.

O sistema CARTOMORPH foi organizado de modo que sua estrutura permita controle sobre os tipos de dados e funções implementados no sistema, facilitando a utilização e implementação de novas funcionalidades. Sendo assim, foram desenvolvidas quatro classes, que juntamente com suas funções, são apresentadas nos próximos tópicos. A divisão da biblioteca de funções do CARTOMORPH nestas quatro classes mencionadas pode ser observada na Figura 6. Cada classe desenvolvida possui um foco de processamento, ou seja, cada uma delas é responsável por conteúdos de processamento distinto, e por esse motivo, muitas vezes há a necessidade de uma classe utilizar funcionalidades implementadas em outras. A Figura 7 apresenta o esquema de uso de funcionalidades entre as classes do sistema desenvolvido.

Vale ressaltar que todas as funções implementadas não alteram o conteúdo da imagem de entrada. Ao invés disso, elas criam uma imagem com os resultados obtidos após cada operação. Esse fato se torna importante para que o resultado do processamento de um *pixel* não influencie o resultado a ser obtido pelos outros *pixels* na continuação da operação.

Figura 6 – Classes de processamento presentes na biblioteca de funções.

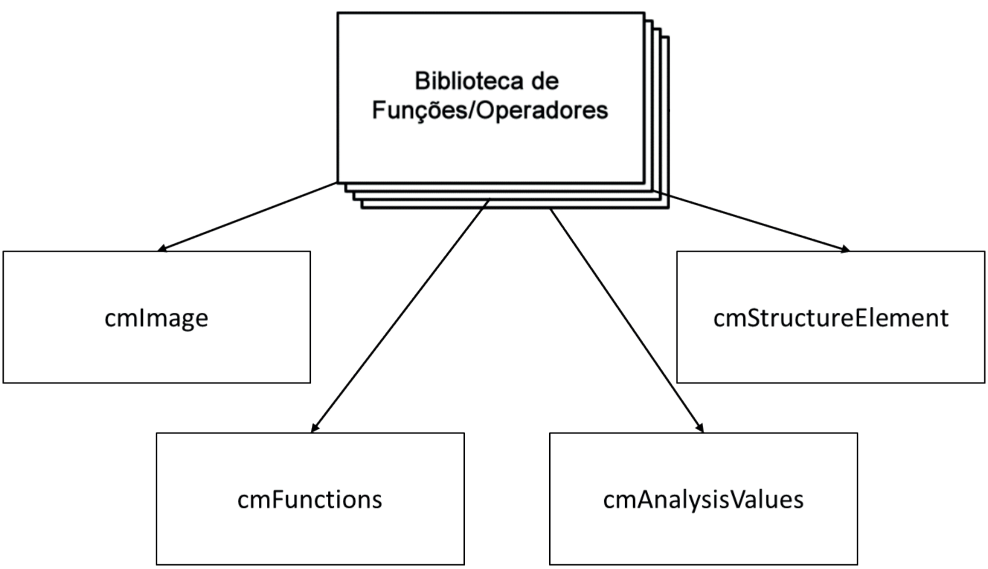
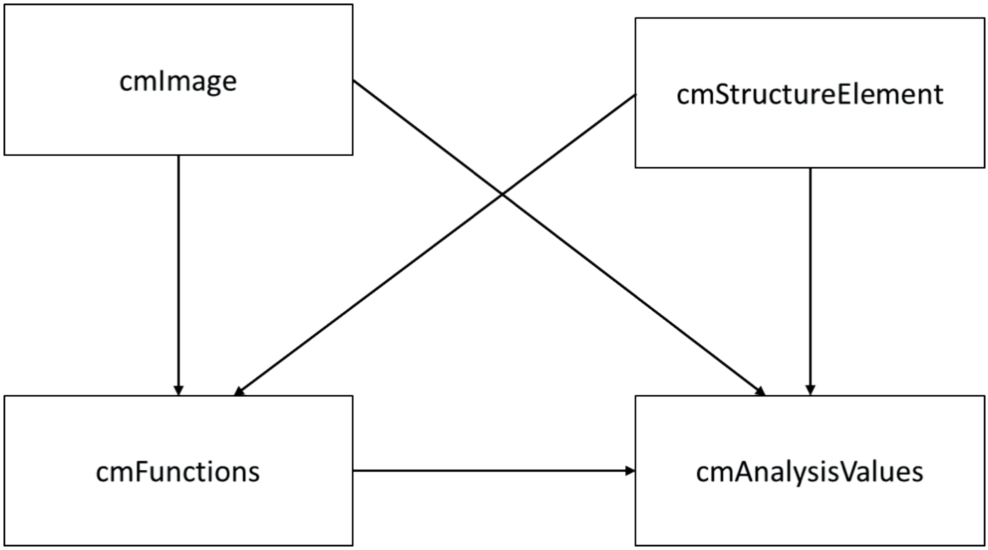


Figura 7 – Esquema de utilização de funções entre as classes.



4.1 CLASSE CMIMAGE

Como primeira etapa no desenvolvimento do sistema, foi necessária a definição de uma estrutura, ou melhor, de uma classe para armazenar a informação contida em uma imagem digital e que permita que essa informação seja acessada e modificada. Nesse sentido, foi criada a classe *cmImage*, a qual possibilita realizar as operações necessárias sobre o conteúdo de uma imagem digital para realizar as operações propostas.

O armazenamento da imagem é feito de modo dinâmico diretamente na memória do computador, sendo que o sistema verifica, ao abrir a imagem, o tipo dela, dentre os disponíveis no sistema, ou seja, se esta é uma imagem binária (*BINARY*), em tons de cinza (*GRAYSCALE*), ou colorida no modo RGB (*RGBCOLOR*). Estes são os tipos de imagens possíveis de serem processadas no sistema.

Além de organizar o armazenamento e o acesso à informação da imagem, essa classe conta com funções que necessitam apenas da própria imagem para serem executadas. Utilizando-se desta classe, a imagem pode ser acessada ou alterada em determinada posição; ser salva para outro arquivo; ter seus níveis digitais expressos em um arquivo texto; convertida entre os tipos de imagens disponíveis; dentre outras aplicações.

Todas as funções presentes na classe *cmImage* disponíveis ao usuário são apresentadas no Quadro 1.

Quadro 1 – Funções disponíveis na classe cmImage.

```
		public:
			cmImage();
			cmImage(int width, int height, int imageType);
			cmImage(char * fileName);
			cmImage(cmImage * orig);
			virtual ~cmImage();
			void cmCopyImage(cmImage * image);
			int cmGetWidth();
			int cmGetHeight();
			bool cmIsRGB();
			bool cmIsGrayScale();
			bool cmIsBinary();
			int cmGetImageType();
			char * cmGetImageTypeTxt();
			int cmGetPixel(int x, int y, int b);
			int cmGetRColor(int x, int y);
			int cmGetGColor(int x, int y);
			int cmGetBColor(int x, int y);
			void cmSetPixel(int x, int y, int b, int value);
			void cmSetPixel(int x, int y, int red, int green, int blue);
			void cmWriteImageToFile(char * fileName);
			void cmSetRColor(int x, int y, int red);
			void cmSetGColor(int x, int y, int green);
			void cmSetBColor(int x, int y, int blue);
			int * cmGetHistogram();
			int * cmGetHistogramPartial(cmImage * imgBin);
			int * cmGetHistogramRelative(int * histogram);
			void cmWriteValuesToTxtFile(char* fileName);
			cmImage * cmRGBToGray();
			cmImage * cmInvertImage();
			cmImage * cmGrayToBinary(int threshold);
			cmImage * cmEqualizeHistogram();
			cmImage * cmGetLayer(int b);
			int cmCountWhitePixels();
			bool cmIsNULL();
```

Muitas das funções presentes nessa classe são funções de atribuição e verificação de valores da intensidade de brilho, dimensões e tipos de imagem, as quais não serão detalhadas. Por outro lado, existem outras funções para abrir, salvar e processar imagens de acordo com determinada técnica de PDI, as quais são apresentadas a seguir.

4.1.1 Abrir um arquivo de imagem no sistema – cmImage

Para carregar no sistema um arquivo de imagem existente, o usuário precisa utilizar a função construtora *cmImage*, para a qual, como parâmetro, deve ser informada uma sequência de caracteres contendo o endereço onde a imagem está armazenada. Esta função é essencial para o sistema, uma vez que ela será utilizada sempre ao abrir uma nova imagem. O sistema utiliza, nesse momento, a biblioteca *EasyBMP* para obter as informações da imagem e converter para o formato definido para o sistema CARTOMORPH. O algoritmo dessa função construtora é apresentado no Quadro 37 (ver Apêndice A), e um

exemplo de utilização pode ser visto no Quadro 2. Neste exemplo, é possível notar que após abrir a imagem no sistema, esta recebe o nome interno *imgOri*, o qual é definido pelo usuário e deve ser utilizado pelo mesmo em funções posteriores.

Quadro 2 – Como Abrir uma Imagem no Sistema.

```
cmImage * imgOri = new cmImage("C:/images/Tiete/original.bmp");
```

4.1.2 Salvar uma imagem em arquivo – cmWriteImageToFile

Uma vez realizadas as operações desejadas, o usuário pode salvar essa imagem novamente em arquivo no computador por meio da função *cmWriteImageToFile*. Como parâmetro da função, o usuário deve informar o caminho completo onde o arquivo de imagem será armazenado. Assim como a função para abrir imagens, essa função utiliza a biblioteca *EasyBMP* para fazer a ligação entre o sistema CARTOMORPH e o armazenamento em disco. O algoritmo dessa função está apresentado no Quadro 38 (ver Apêndice A), e o modo de uso no Quadro 3.

Quadro 3 – Como Salvar uma Imagem do Sistema no Disco.

```
imgResult->cmWriteImageToFile("C:/images/Tiete/invertida.bmp");
```

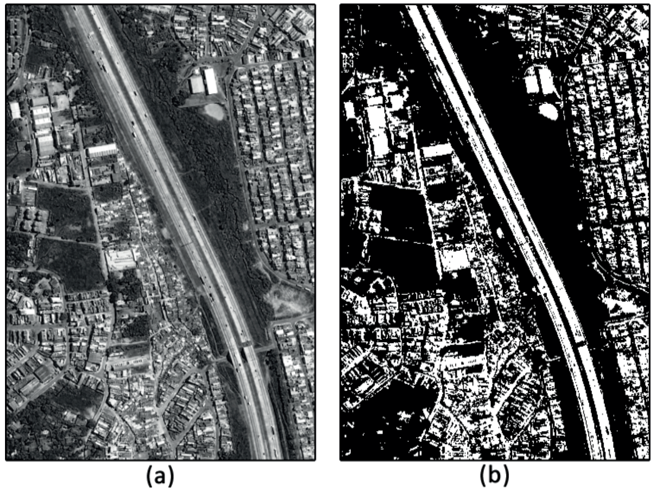
4.1.3 Binarizar uma Imagem - cmGrayToBinary

A função de binarização, descrita na Seção 2.2.1, foi implementada no sistema CARTOMORPH com o nome *cmGrayToBinary*. Como parâmetro, a função necessita de um valor inteiro de limiar repassado para a função por meio da variável *threshold*. O Quadro 39 (ver Apêndice A) apresenta o algoritmo desta função, enquanto o Quadro 4 exemplifica o uso desta função, sendo o resultado da operação, com limiar 140, apresentado na Figura 8.

Quadro 4 – Como Aplicar a Binarização.

```
cmImage * imgBin = img->cmGrayTobinary(140);
```

Figura 8 - Exemplo de Binarização. (a) Imagem Original (b) Resultado da Binarização.



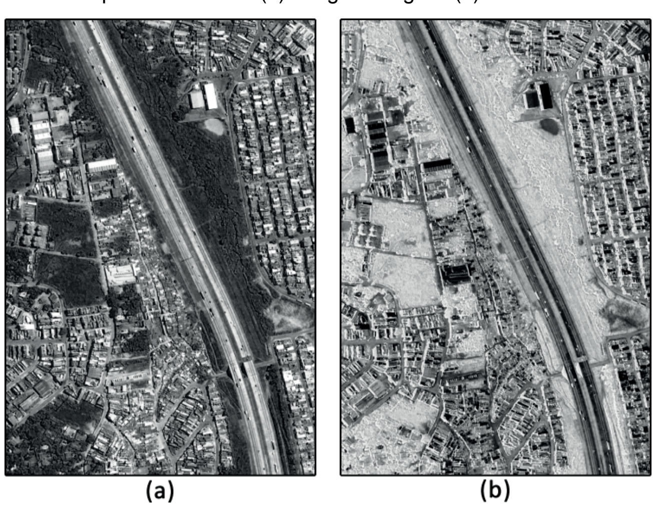
4.1.4 Inverter uma Imagem - cmInvertImage

A Seção 2.2.2 apresenta os conceitos referentes à função de inversão, sendo que no sistema CARTOMORPH essa função foi denominada *cmInvertImage* e está implementada na classe *cmImage*. Essa função não necessita de parâmetros. O Quadro 40 apresenta o algoritmo desenvolvido para realizar essa operação, enquanto o modo de uso da mesma é exemplificado no Quadro 5 e o resultado obtido é apresentado na Figura 9.

Quadro 5 – Como Aplicar a Inversão.

```
cmImage *imgResult = imgOri->cmInvertImage();
```

Figura 9 - Exemplo de Inversão. (a) Imagem Original (b) Resultado da Inversão.



4.1.5 Converter uma Imagem RGB para Tons de Cinza – cmRGBToGray

Para converter uma imagem colorida RGB, a qual possui três camadas de cores, para uma imagem em tons de cinza, que possui apenas uma camada, deve-se dar atenção ao cálculo a ser realizado para este procedimento. Na literatura é possível encontrar diferentes formas de realizar esta conversão. Dentre as opções encontradas, a função de conversão cmRGBToGray, presente no sistema desenvolvido utiliza, o cálculo da Luminância a partir das três camadas existentes para encontrar o nível de cinza que a imagem resultante deve possuir. O cálculo de Luminância refere-se a uma quantidade não linear utilizada para representar o brilho em um sistema de vídeo, e de acordo com a recomendação ITU-R BT.601-4 pode ser definido de acordo com a Equação (27) (INTERNATIONAL TELECOMMUNICATION UNION, 1994).

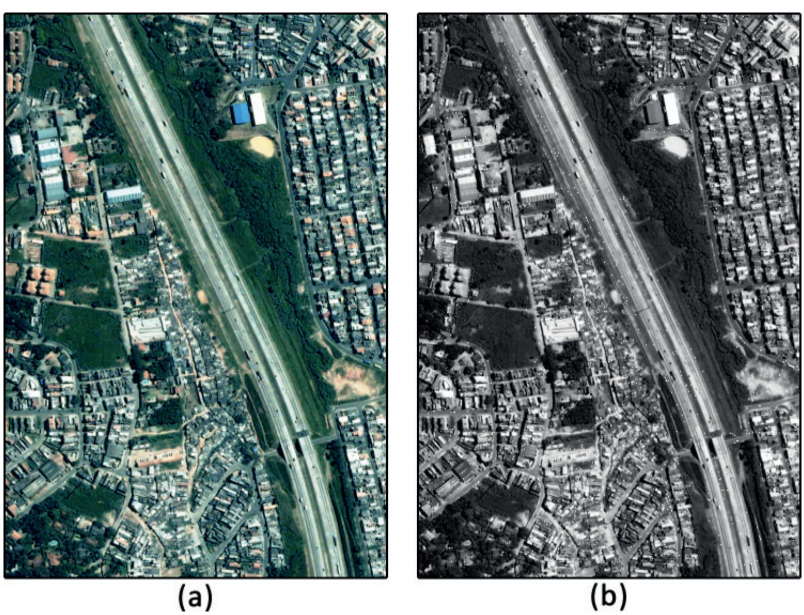
$$Y = 0{,}299 * R + 0{,}587 * \mathrm{G} + 0{,}114 * \mathrm{B} \tag{27}$$

Dessa forma, a função encontra o valor do nível de cinza da imagem resultante com base nos valores das três camadas existentes em uma imagem colorida RGB. O Quadro 41 (ver Apêndice A) apresenta o algoritmo desenvolvido para esta função enquanto que o Quadro 6 exemplifica o uso desta função, sendo o resultado apresentado na Figura 10.

Quadro 6 – Como Aplicar a Conversão para Escala de Cinzas.

```
cmImage * imgGray = img->cmRGBToGray();
```

Figura 10 – Exemplo de Conversão RGB para Tons de Cinza. (a) Imagem Original (b) Resultado da Conversão para Cinza.



4.1.6 Equalização de Histogramas

O objetivo da função de Equalização de Histograma consiste em obter estatisticamente uma melhor distribuição para os valores da imagem. Com este intuito, cada *pixel* da imagem é analisado referente aos valores de frequência acumulada, como apresentado na Equação (28).

$$valor_equalizado = \max\left[0, round\left(\frac{total_de_níveis * frequência_acumulada}{qtd_de_colunas * qtd_de_linhas}\right) - 1\right] \quad (28)$$

Com base nesta equação é possível distribuir os níveis de uma imagem criando uma nova com uma distribuição igualitária de valores. O Quadro 42 (ver Apêndice A) apresenta o algoritmo desenvolvido para esta função. O Quadro 7 demonstra como aplicar essa operação em uma imagem, enquanto a Figura 11 apresenta o resultado obtido por meio da aplicação dessa função.

Quadro 7 – Como Aplicar a Equalização de Histograma.

```
cmImage * imgResult = img->cmEqualizeHistogram();
```

Figura 11 – Exemplo de Equalização de Histograma. (a) Imagem Original (b) Resultado da Equalização de Histograma.



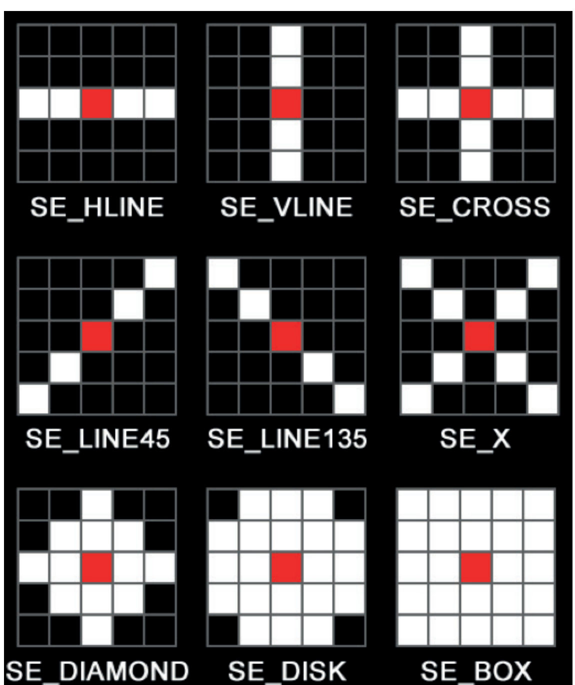
(a) (b)

4.2 CLASSE CMSTRUCTUREELEMENT

Tendo como foco a teoria da morfologia matemática, um conceito importante dentro desse tema é o elemento estruturante (EE). Praticamente todas as operações relacionadas com a teoria da MM utilizam um elemento estruturante em seus cálculo como apresentado na Seção 2.3. Dessa forma, verificou-se a necessidade do desenvolvimento de uma classe específica para essas estruturas, a qual foi denominada de *cmStructureElement*. Esta classe possui as funções de criação, conhecidas como construtoras; as funções de acesso ao conteúdo do EE; e duas funções de processamento do conteúdo do EE, como pode ser visualizado no Quadro 8. Para processamento do EE temos apenas uma função para criar um EE transposto e para girar o mesmo em 90º. Por serem funções matemáticas bem definidas na literatura, o desenvolvimento dessas funções não será detalhado neste trabalho.

O usuário pode criar um elemento estruturante de quatro maneiras. A primeira consiste em um elemento estruturante vazio (todos os valores iguais a zero), para o qual o usuário informa apenas as dimensões dele. A segunda opção consiste em criar um EE informando as dimensões do mesmo e o tipo que este terá. O sistema CARTOMORPH dispõe de 9 tipos de EE pré-definidos. São eles: linha horizontal (*SE_HLINE*); linha vertical (*SE_VLINE*); cruz (*SE_CROSS*); linha diagonal em 45º (*SE_LINE45*); linha diagonal em 135º (*SE_LINE135*); cruz inclinada em X (*SE_X*); diamante (*SE_DIAMOND*); disco (*SE_DISK*); e caixa (*SE_BOX*). Estes diferentes tipos de EE estão exemplificados na Figura 12 com dimensões 5x5 e ponto origem representado na cor vermelha.

Figura 12 - Elementos Estruturantes Disponíveis no CARTOMORPH.



Como terceira opção na criação de um EE, o usuário pode definir, com os conceitos de programação em C/C++, um vetor de números inteiros contendo a disposição do EE que deseja e informar ao sistema esse vetor por meio de um parâmetro. Para finalizar, a última opção do usuário consiste em informar ao sistema outro EE já criado para que seja criada uma cópia dele. Todas essas opções são executadas por meio da função construtora da classe *cmStructureElement*, e podem ser visualizadas no Quadro 8.

Quadro 8 – Funções Disponíveis na Classe cmStructureElement.

```
public:
    cmStructureElement(int width, int height);
    cmStructureElement(int width, int height, int typeSE);
    cmStructureElement(int width, int height, int * se_values);
    cmStructureElement(const cmStructureElement& orig);
    virtual ~cmStructureElement();
    int cmGetWidth();
    int cmGetHeight();
    int cmGetValue(int x, int y);
    cmStructureElement * cmTranspose();
    cmStructureElement * cmRotate();
```

Para exemplificar a criação de um elemento estruturante, o Quadro 9 apresenta a chamada que deve ser realizada para a criação de um EE de dimensões 7x7 do tipo disco.

Quadro 9 – Como Criar um Elemento Estruturante.

```
cmStructureElement * se = new cmStructureElement(7,7,SE_DISK);
```

4.3 CLASSE CMFUNCTIONS

A classe *cmFunctions* foi desenvolvida para conter todas as operações que envolvam não apenas uma imagem, mas várias delas, ou imagens em conjunto com elementos estruturantes. Sendo assim, a classe *cmFunctions* não realiza o armazenamento de informações, apenas processa informações contidas em imagens e/ou elementos estruturantes retornando o resultado para o usuário. Dessa forma todas as funções presentes nessa classe são do tipo estática, uma vez que não operam sobre informações presentes em objetos da mesma classe. As funções implementadas nessa classe serão detalhadas nas próximas seções.

4.3.1 Filtro da Média

O filtro da média, definido na Seção 2.2.3, é um operador de suavização da imagem, sendo muito utilizado para atenuação de ruídos. Dentro do sistema, essa função foi implementada na classe *cmFunctions* e denominada *cmFilterAVG*. Por utilizar a vizinhança de cada *pixel* durante as operações, o algoritmo faz uso de um EE para realizar os cálculos na vizinhança desejada pelo usuário. Além disso, o usuário necessita informar qual a imagem a ser processada por meio de parâmetros repassados à função. O algoritmo desenvolvido é apresentado no Apêndice A pelo Quadro 43, tendo sua utilização exemplificada no Quadro 10 e um exemplo de resultado demonstrado na Figura 13.

Quadro 10 – Como Aplicar o Filtro da Média.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmFilterAVG(img,se);
```

Figura 13- Exemplo de Filtro da Média. (a) Imagem Original (b) Resultado do Filtro da Média.



(a)



(b)

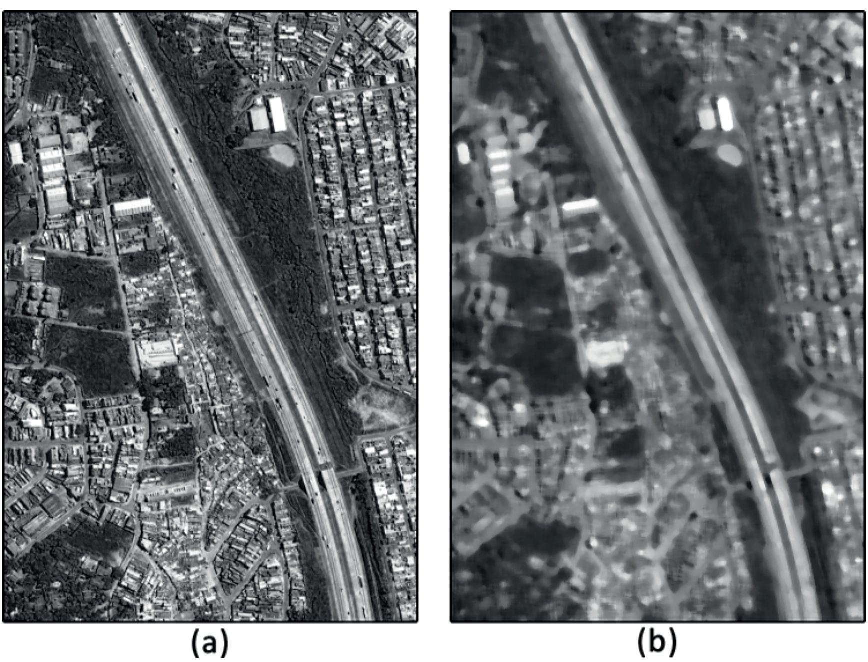
4.3.2 Filtro da Mediana

O filtro da mediana, assim como o da média, foi implementado na classe *cmFunctions* recebendo o nome de *cmFilterMedian*. O algoritmo necessita que o usuário informe a imagem que será filtrada e o elemento estruturante que será utilizado como base de vizinhança, como foi definida a função na Seção 2.2.3. Esse filtro é muito utilizado para realizar remoções de ruídos pontuais. O algoritmo desenvolvido é apresentado no Quadro 44 (ver Apêndice A). O Quadro 11 apresenta o modo de uso dessa função, enquanto a Figura 14 ilustra um resultado obtido.

Quadro 11 – Como Aplicar o Filtro da Mediana.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmFilterMedian(img,se);
```

Figura 14 - Exemplo de Filtro da Mediana. (a) Imagem Original (b) Resultado do Filtro da Mediana.



4.3.3 Filtro Laplaciano

O filtro Laplaciano como detector de bordas, definido pela Seção 2.2.4, foi implementado no CARTOMORPH na classe *cmFunctions* recebendo o nome de *cmFilterEdgesLaplace*. Esse filtro possui a vizinhança definida pelo conceito, portanto o usuário não necessita informar nenhum EE, apenas a imagem que passará por esse processo. O algoritmo desenvolvido para essa função é apresentado no Apêndice A no Quadro 45, enquanto o Quadro 12 apresenta o seu uso e o resultado obtido como exemplo está apresentado na Figura 15.

Quadro 12 – Como Filtrar uma Imagem por Laplace.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmImage * imgResult = cmFunctions::cmFilterEdgesLaplace(img);
```

Figura 15 - Exemplo de Filtro Laplaciano. (a) Imagem Original (b) Imagem Filtrada.



4.3.4 Erosão

A função de erosão, juntamente com a dilatação, é uma das funções básicas da M nforme visto nas Seções 2.3.3 e 2.3.4. Foi implementada na classe *cmFunctions* e denominada *cmErode,* sendo necessário que o usuário informe dois parâmetros de entrada, a imagem a ser erodida e o elemento estruturante que será utilizado durante o processo. O algoritmo desenvolvido é capaz de realizar a operação em imagens binárias, imagens em tons de cinza e imagens coloridas do tipo RGB. Para efeito de exemplificação o Quadro 46 (ver Apêndice A) apresenta um algoritmo reduzido que funciona em imagens binárias ou em tons de cinza. No algoritmo é possível observar que as linhas 35, 41, 45 e 48 são comandos que verificam se o EE está situado nas bordas da imagem impedindo que o algoritmo busque por informações fora dos limites da imagem. Dessa forma, o algoritmo cria a imagem resultante com as mesmas dimensões da imagem original. Contudo, vale ressaltar que dessa forma as bordas da imagem resultante sofreram uma menor influência por parte do elemento estruturante do que os elementos centrais da mesma.

O Quadro 13 exemplifica o uso dessa função, sendo o resultado da operação apresentado na Figura 16.

Quadro 13 – Como Aplicar a Erosão.

```
cmImage * img = new cmImage("images/Qualificacao/Original.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmErode(img,se);
```

Figura 16 - Exemplo de Erosão. (a) Imagem Original (b) Resultado da Erosão.



4.3.5 Dilatação

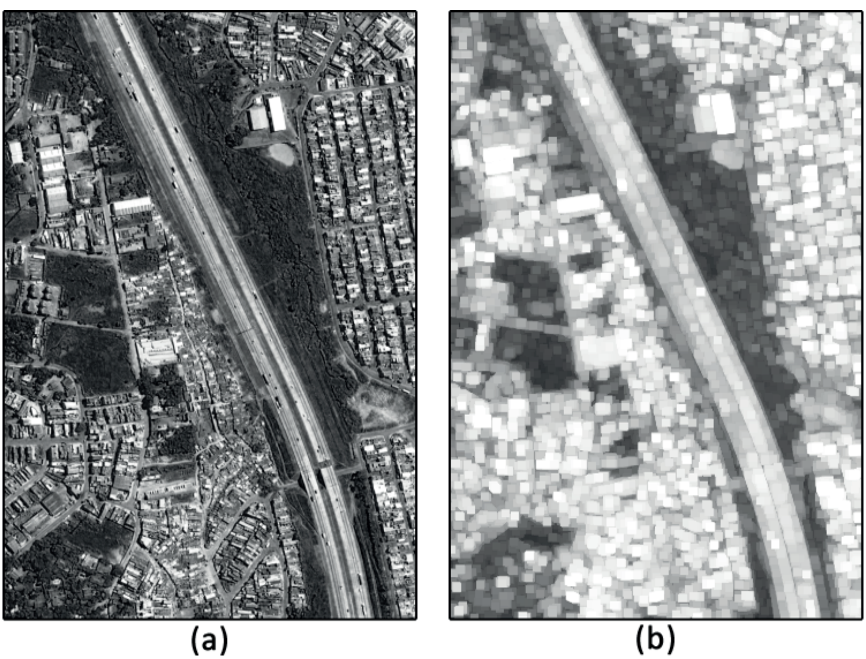
A função de dilatação possui as mesmas características da função de erosão, tendo sido implementada na classe *cmFunctions* e recebendo o nome de *cmDilate* o definido pelas Seções 2.3.3 e 2.3.4, o algoritmo implementado é capaz de realizar essa operação sobre imagens binárias, em tons de cinza ou coloridas RGB. Para exemplificação, o Quadro 47 (ver Apêndice A) apresenta o algoritmo simplificado que funciona sobre imagens binárias e em tons de cinza. Do mesmo modo que a função de erosão, a função de dilatação evita a busca de informações além dos limites da imagem quando o EE está situado nas bordas desta.

Para exemplificar o uso da função de dilatação, o Quadro 14 apresenta a chamada a esta função com os resultados obtidos ilustrados na Figura 17.

Quadro 14 – Como Aplicar a Dilatação.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmDilate(img,se);
```

Figura 17 - Exemplo de Dilatação. (a) Imagem Original (b) Resultado da Dilatação.



4.3.6 Abertura

A função de abertura, definida na Seção 2.3.5, foi implementada na classe *cmFunctions*, sendo necessário que o usuário passe, por meio dos parâmetros, qual imagem será processada e qual EE será utilizado. O Quadro 48 (ver Apêndice A) apresenta o algoritmo desenvolvido para esta função, enquanto o Quadro 15 exemplifica o uso e a Figura 18 apresenta o resultado obtido com a aplicação dessa função.

Quadro 15 – Como Aplicar a Abertura Morfológica.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmOpen(img,se);
```

Figura 18 - Exemplo de Abertura. (a) Imagem Original (b) Resultado da Abertura.



(a) (b)

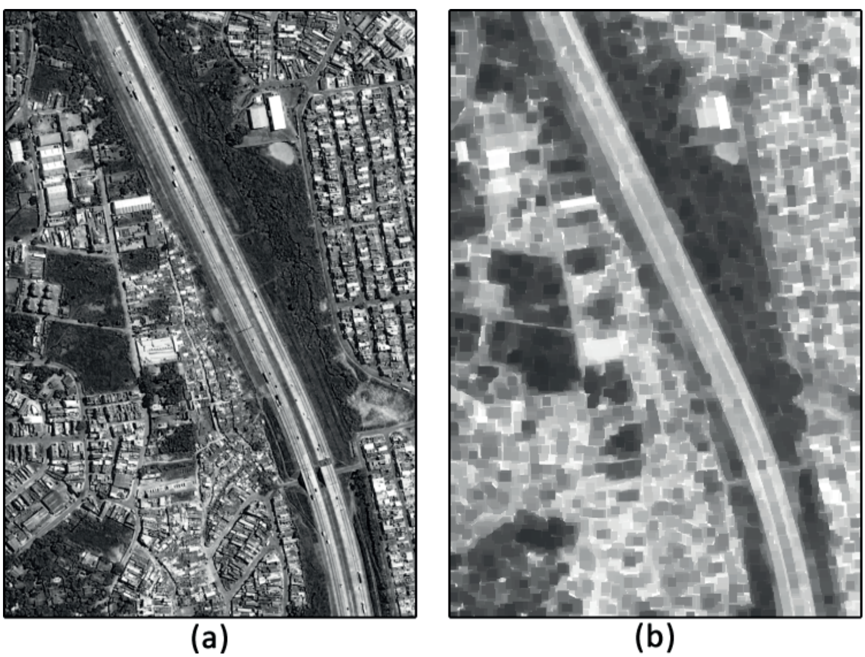
4.3.7 Fechamento

O operador de fechamento, definido na Seção 2.3.5, foi denominado de *cmClose* e implementado do mesmo modo como o operador de abertura, ou seja, foi implementado na classe *cmFunctions*, sendo o usuário responsável por informar qual a imagem a ser processada e o EE a ser utilizado pela função. O Quadro 49 (ver Apêndice A) apresenta o algoritmo desenvolvido para esta função, enquanto o Quadro 16 exemplifica o uso e a Figura 19 apresenta o resultado obtido com a aplicação do fechamento.

Quadro 16 – Como Aplicar o Fechamento Morfológico.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmClose(img,se);
```

Figura 19 - Exemplo de Fechamento. (a) Imagem Original (b) Resultado do Fechamento.



4.3.8 Gradiente da Erosão

O gradiente da erosão funciona como um operador de detecção de bordas como definido na Seção 2.3.7. Esta Função foi implementada na classe *cmFunctions,* sendo denominada de *cmGradientInternal*. Como a maioria das outras funções, o usuário necessita informar ao sistema qual a variável correspondente à imagem original e qual o EE a ser utilizado. Por se tratar de uma junção de outras funções, a implementação do gradiente interno pode ser simplificada como apresentado no Apêndice A pelo Quadro 50. O Quadro 17 demonstra como esta função deve ser utilizada, enquanto a Figura 20 ilustra um resultado obtido pela mesma.

Quadro 17 – Como Aplicar o Gradiente da Erosão.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGradientInternal(img,se);
```

Figura 20 - Exemplo de Gradiente da Erosão. (a) Imagem Original (b) Resultado do Gradiente da Erosão.





(a) (b)

4.3.9 Gradiente da Dilatação

Implementado na classe *cmFunctions*, o gradiente da dilatação também tem como característica a detecção das bordas dos alvos presentes na imagem, no entanto, ao contrário do gradiente da erosão, o gradiente externo detecta as bordas externas dos alvos. Essa função foi denominada no sistema *cmGradientExternal*. Por ser caracterizado pela união de outras funções, o algoritmo dessa função pode ser simplificado como apresentado no Apêndice A pelo Quadro 51. O uso dessa função é apresentado no Quadro 18 e o resultado obtido pela mesma exemplificado na Figura 21.

Quadro 18 – Como Aplicar o Gradiente da Dilatação.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGradientExternal(img,se);
```

Figura 21 - Exemplo de Gradiente da Dilatação. (a) Imagem Original (b) Resultado do Gradiente da Dilatação.



4.3.10 Gradiente Total

O gradiente tot omo definido pela Seção 2.3.7, pode ser considerado a soma dos gradientes da erosão e da dilatação. Essa função é caracterizada por detectar tanto as bordas internas como as bordas externas dos alvos presente na imagem. Assim como os gradientes interno e externo, essa função está implementada na classe *cmFunctions* e pode ser descrita a partir de outras funções já mencionadas, sendo assim, seu algoritmo pode ser simplificado como apresentado no Apêndice A pelo Quadro 52. No sistema, essa função recebeu o nome de *cmGradientTotal* e seu uso é demonstrado pelo Quadro 19, enquanto a Figura 22 exemplifica um resultado obtido por essa função.

Quadro 19 – Como Aplicar o Gradiente Total.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGradientTotal(img,se);
```

Figura 22 - Exemplo de Gradiente Total. (a) Imagem Original (b) Resultado do Gradiente Total.



(a) (b)

4.3.11 Combinado Mínimo (GMin)

A função do combinado mínimo, ou GMin, recebeu o nome de *cmGMin,* sendo implementada na classe *cmFunctions*. Para sua correta execução, o usuário deve fornecer a imagem a ser processada e o EE a ser utilizado por meio dos parâmetros da função. Por ser uma combinação de outras funções, esse algoritmo pode ser simplificado como apresentado no Apêndice A pelo Quadro 53. O Quadro 20 demonstra o modo de uso dessa função, enquanto a Figura 23 apresenta o resultado obtido.

Quadro 20 – Como Aplicar a Função do Combinado Mínimo.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGMin(img,se);
```

Figura 23 - Exemplo de Combinado Mínimo. (a) Imagem Original (b) Resultado do Combinado Mínimo.



(a) (b)

4.3.12 Combinado Máximo (GMax)

O combinado má foi definido pela Seção 2.3.7 e implementado com o nome de *cmGMax*, sendo o algoritmo apresentado no Apêndice A pelo Quadro 54. Para utilizar essa função, o usuário precisa informar, por parâmetros, qual imagem será processada e qual EE será utilizado durante o processo, como pode ser visto no Quadro 21. Um exemplo de resultado obtido com essa função pode ser visualizado na Figura 24.

Quadro 21 – Como Aplicar a Função do Combinado Máximo.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGMax(img,se);
```

Figura 24 - Exemplo de Combinado Máximo. (a) Imagem Original (b) Resultado do Combinado Máximo.



(a)



(b)

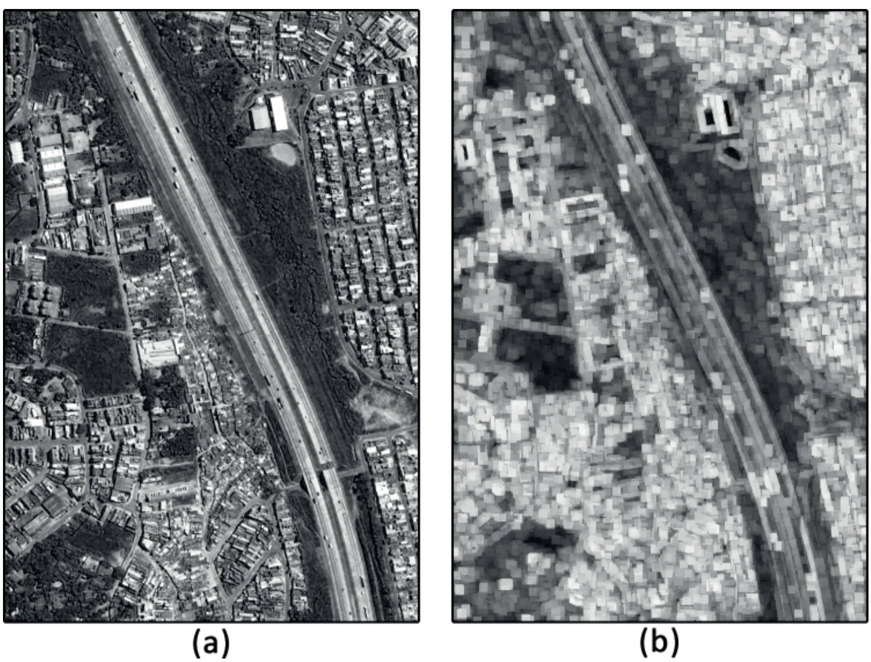
4.3.13 Combinado da Soma (GSum)

A função do combinado da soma foi denominada no sistema por *cmGSum*, sendo o usuário responsável por informar o EE e a imagem que será processada, por meio dos parâme tros. Como definido na Seção 2.3.7, o combinado da soma é caracterizado por outras funções previamente descritas, sendo assim, o algoritmo dessa função foi implementado como apresentado no Apêndice A pelo Quadro 55. O uso dessa função está exemplificado no Quadro 22 e a Figura 25 apresenta o resultado obtido com essa operação.

Quadro 22 – Como Aplicar a Função do Combinado da Soma.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGSum(img,se);
```

Figura 25 - Exemplo de Combinado da Soma. (a) Imagem Original (b) Resultado do Combinado da Soma.



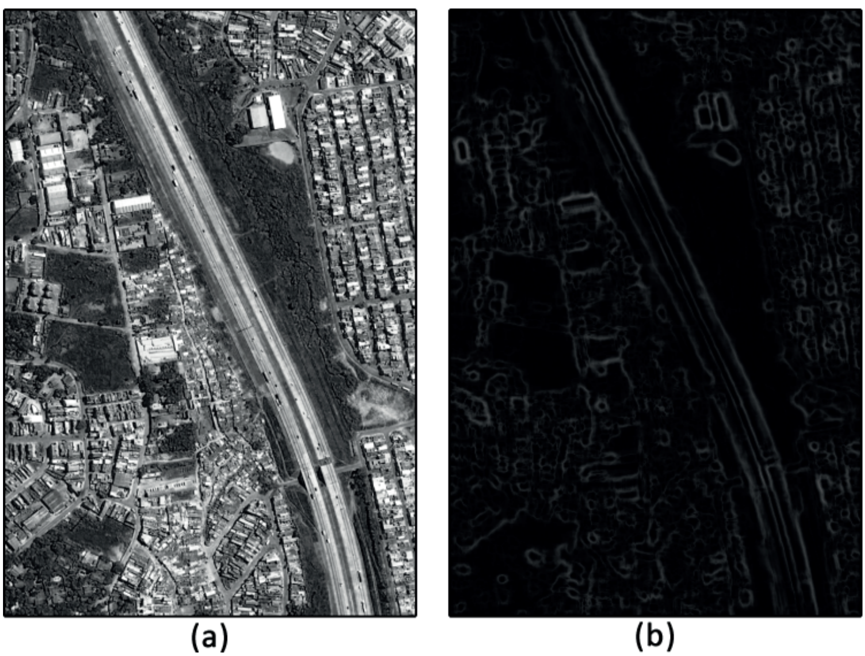
4.3.14 Combinado de Borramento Mínimo (GBlur)

E ssa função, definida na Seção 2.3.7, foi implementada no sistema com o nome de *cmGBlur*. Assim como as funções de gradientes apresentadas anteriormente, o usuário necessita informar a imagem e o EE que serão utilizados no processamento. A primeira etapa dessa operação consiste em realizar um filtro de suavização, sendo assim, o filtro da média foi utilizado, como pode ser observado no algoritmo apresentado no Apêndice A pelo Quadro 56. O modo de uso dessa função está demonstrado no Quadro 23 e o resultado obtido é exemplificado na Figura 26.

Quadro 23 – Como Aplicar o Combinado de Borramento Mínimo.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmGBlur(img,se);
```

Figura 26 - Exemplo de Combinado de Borramento Mínimo. (a) Imagem Original (b) Resultado do Combinado de Borramento Mínimo.



4.3.15 Top-hat por Abertura

O operador de *top-hat* por abertura foi definido na Seção 2.3.6 e implementado no sistema pela função denominada *cmTophatOpen*. Como a maior parte das outras funções, o *top-hat* por abertura necessita da imagem que será processada e do EE que será utilizado, os quais devem ser informados por parâmetros para a função. O Quadro 57 (ver Apêndice A) apresenta o algoritmo implementado para essa função, enquanto o Quadro 24 apresenta seu modo de uso. Um exemplo de resultado obtido com o uso dessa função pode ser visualizado na Figura 27.

Quadro 24 – Como Aplicar a Função de Top-hat por Abertura.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmTophatOpen(img,se);
```

Figura 27 - Exemplo de Top-hat por Abertura. (a) Imagem Original (b) Resultado do Top-hat por Abertura.



(a) (b)

4.3.16 Top-hat por Fechamento

Assim como o *top-hat* por abertura, esse operador está definido na Seção 2.3.6 e foi implementado no sistema com o nome de *cmTophatClose*. Para executar essa função, o usuário necessita informar, por parâmetros, o EE e a imagem que serão utilizados para o processamento. O algoritmo dessa função é apresentado no Apêndice A pelo Quadro 58 e o modo de uso no Quadro 25. Um exemplo de resultado obtido com essa função é visualizado na Figura 28.

Quadro 25 – Como Aplicar a Função de Top-har por fechamento.

```
cmImage * img = new cmImage("images/Qualificacao/negOriginal.bmp");
cmStructureElement * se = new cmStructureElement(7,7,SE_BOX);
cmImage * imgResult = cmFunctions::cmTophatClose(img,se);
```

Figura 28 - Exemplo de Top-hat por Fechamento. (a) Imagem Original (b) Resultado do Top-hat por Fechamento.

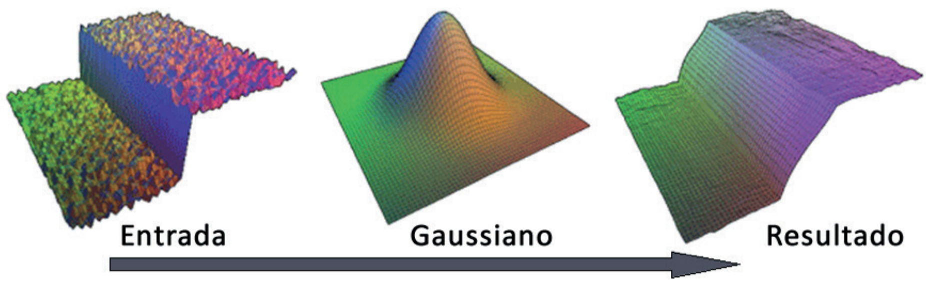


(a) (b)

4.3.17 Filtro Gaussiano

O filtro Gaussiano é uma função que utiliza diferentes pesos para a vizinhança de um *pixel* de acordo com a distância do ponto vizinho para o ponto que receberá o novo valor. Dessa forma, o filtro Gaussiano suaviza a imagem considerando que os *pixels* mais próximos ao *pixel* de interesse devem ter maior influência sobre o novo valor do que os *pixels* mais distantes. Pode-se observar ao centro da Figura 29 o formato que assume uma máscara de um Filtro Gaussiano, além de verificar um exemplo de resultado obtido por um filtro Gaussiano para determinada entrada de dados.

Figura 29 – Exemplo de Filtro Gaussiano.



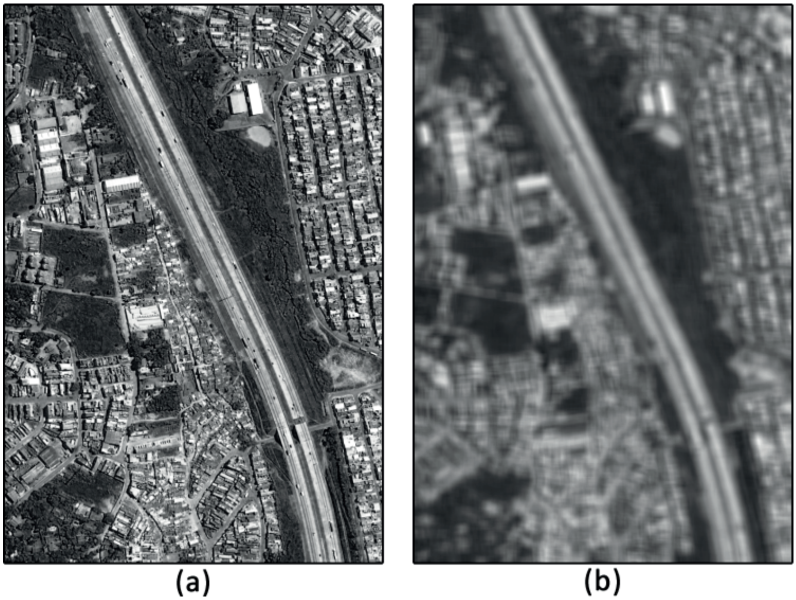
Fonte: Adaptado de (DURAND; DORSEY, 2002)

Dessa forma, o filtro Gaussiano tem a propriedade de suavizar a imagem considerando os *pixels* mais próximos como mais influentes no resultado. Para executar o filtro Gaussiano há a necessidade de informar um desvio padrão, seguindo as equações matemáticas da função Gaussiana, e a dimensão da vizinhança a ser utilizada. Essa função foi implementada com o nome de *cmGaussianFilter* e o algoritmo desenvolvido é apresentado no Apêndice A pelo Quadro 59. O Quadro 26 apresenta como utilizar essa função enquanto um resultado obtido com essa função é exemplificado na Figura 30.

Quadro 26 – Como Aplicar o Filtro Gaussiano.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/ori_b.bmp");
cmImage * imgResult = cmFunctions::cmGaussianFilter(imgOri,9,40);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/gaussian.bmp");
```

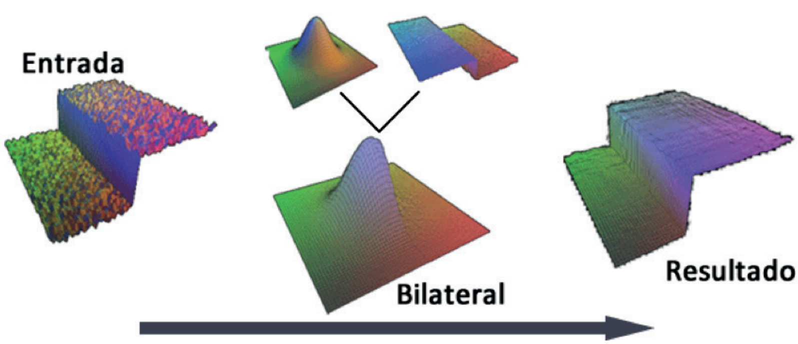
Figura 30 – Exemplo de Filtro Gaussiano. (a) Imagem Original (b) Resultado do Filtro Gaussiano.



4.3.18 Filtro Bilateral

O filtro Bilateral utiliza os conceitos do filtro Gaussiano para suavizar a imagem atribuindo maior influência no cálculo ao valor dos *pixels* mais próximos ao *pixel* de interesse. No entanto, o filtro bilateral possui uma característica extra de preservar as bordas dos alvos presentes na imagem. Tal característica se torna possível ao mesclar os pesos atribuídos pelo filtro Gaussiano com a intensidade dos valores presentes em cada *pixel* de determinada vizinhança. A Figura 31 apresenta um exemplo da função Bilateral, na qual; no seu centro, é possível observar que a união do filtro Gaussiano com a grande diferença de valores nas bordas do alvo gera um novo filtro que suaviza a imagem, mas que consegue preservar a borda do alvo presente na imagem.

Figura 31 – Exemplo de Filtro Bilateral.



Fonte: Adaptado de (DURAND; DORSEY, 2002)

Dessa forma, o filtro Bilateral consegue manter as bordas dos alvos presentes na imagem suavizando-a de tal forma que um determinado *pixel* tenha maior influência dos *pixels* mais próximos a ele. Para a aplicação desse filtro, é necessário informar a dimensão da região a ser considerada, assim como o valor dos desvios padrões necessários para a realização dos cálculos Gaussiano e Bilateral. O Quadro 60 (ver Apêndice A) apresenta o algoritmo desenvolvido para esta função enquanto o Quadro 27 e a Figura 32 apresentam, respectivamente, o modo de uso e o exemplo obtido com a aplicação dessa função.

Quadro 27 – Como Aplicar o Filtro Bilateral.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/ori_b.bmp");
cmImage * imgResult = cmFunctions::cmBilateralFilter(imgOri,9,40,40);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/gaussian_bilateral.bmp");
```

Figura 32 – Exemplo de Filtro Bilateral. (a) Imagem Original (b) Resultado do Filtro Bilateral.

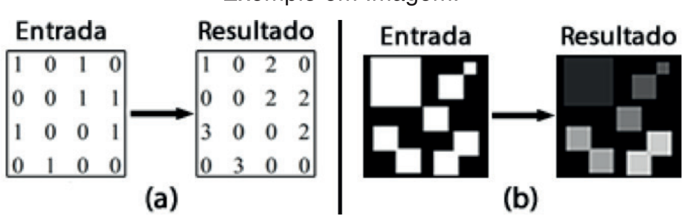


(a) (b)

4.3.19 Rotulação de Alvos

A rotulação de alvos consiste em fazer com que componentes não conectados recebam diferentes valores, os quais definiriam o rótulo de cada alvo presente na imagem. Cada alvo presente na imagem recebe um valor diferente de rótulo possibilitando que cada um seja identificado separadamente. Essa função funciona somente sobre imagens binárias, nas quais é possível identificar os diferentes alvos, e necessita como parâmetros, além da imagem: um elemento estruturante que definirá a conectividade a ser utilizada para separar os alvos presentes; a cor de interesse dos alvos a serem rotulados (branco ou preto – WHITE ou BLACK); e o endereço de uma variável inteira, a qual funciona como parâmetro de retorno para a quantidade de alvos rotulados. No sistema foram implementadas três possíveis funções que realizam esta tarefa, *cmLabel*, *cmLabelImg* e *cmLabelImgEq*. As três funções necessitam dos mesmos parâmetros descritos anteriormente, o que as diferenciam é o modo de apresentar, ou retornar, o resultado. A primeira delas retorna uma matriz numérica com todos os alvos devidamente rotulados. A segunda, *cmLabelImg*, retorna a matriz numérica em forma de imagem, enquanto a última função, *cmLabelImgEq*, retorna também uma imagem, porém equalizada para melhor diferenciação dos alvos. Contudo, determinada imagem pode possuir uma quantidade de alvos maior do que o número de níveis de cinza disponíveis em uma imagem em tons de cinza (256 cores), e por este motivo pode ocorrer dessa imagem possuir alvos com a mesma tonalidade. Para resolver esse problema, a solução é utilizar a função *cmLabel* que retorna a matriz numérica e possibilita uma quantidade superior de diferentes rótulos. Para exemplificar o funcionamento desta função, a Figura 33 apresenta o resultado a ser obtido com a aplicação desta função em uma matriz, Figura 33 (a), e em uma imagem sintética, Figura 33 (b), ambas utilizando um elemento estruturante do tipo caixa para definir a conectividade entre alvos.

Figura 33 – Exemplos Sintéticos da Função de Rotulação de Imagem. (a) Exemplo em Matiz (b) Exemplo em Imagem.

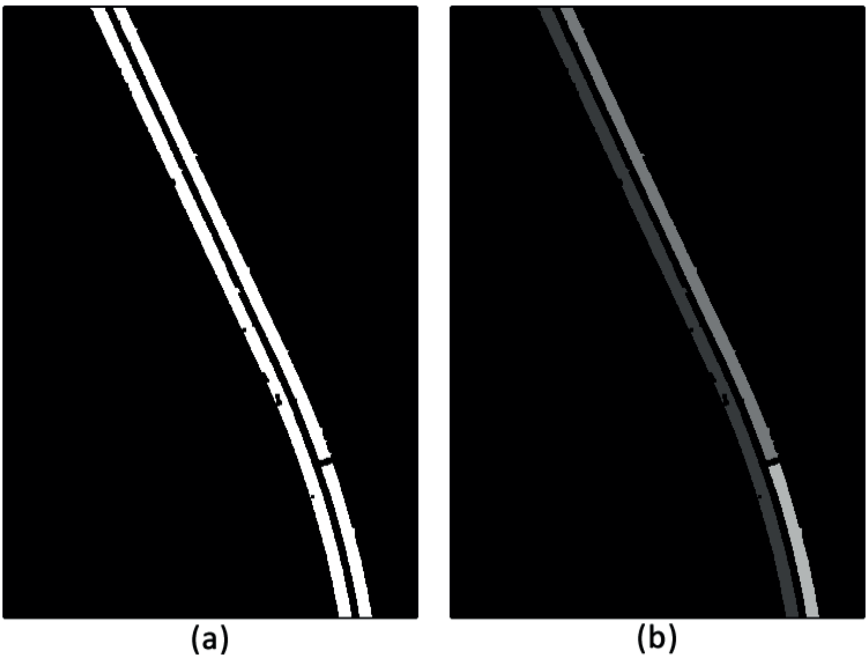


O algoritmo da função *cmLabel* está apresentado no Apêndice A pelo Quadro 61, enquanto o Quadro 28 demonstra o modo de uso da função *cmLabelImg,* tendo um exemplo de resultado real apresentado na Figura 34.

Quadro 28 – Como Aplicar a Rotulação de Imagens.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/detectada.bmp");
cmStructureElement * se = new cmStructureElement(3,3,SE_BOX);
int qtdLabel;
cmImage * imgResult = cmFunctions::cmLabelImgEq(imgOri,se,WHITE,&qtdLabel);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/rotulada.bmp");
```

Figura 34 – Exemplo de Rotulação de Imagens. (a) Imagem de Entrada (b) Resultado da Rotulação.



4.3.20 Abertura e Fechamento por Área.

As funções de Abertura e Fechamento por Área são muito similares e por este motivo serão demonstradas em um único tópico. O objetivo da função de Abertura por Área consiste em eliminar pequenos alvos, que não sejam de interesse presentes na imagem. De modo oposto, a função de Fechamento por Área tem por objetivo incorporar pequenas áreas, que estão internas a determinado alvo, mas que não pertencem a este. Essas funções necessitam como parâmetros a imagem de entrada, a qual necessita ser binária; um elemento estruturante, o qual será utilizado para determinar a conectividade dos alvos; e um valor máximo da área, em quantidade de *pixels*, que deve ser removida ou incorporada a imagem. Para exemplificar, a Figura 35 apresenta o resultado de ambas as operações em uma matriz, enquanto a Figura 36 apresenta os resultados da aplicação em uma imagem sintética.

Figura 35 – Exemplo de Abertura e Fechamento por Área em uma Matriz. (a) Matriz de Entrada (b) Resultado da Abertura por Área (c) Resultado do Fechamento por Área.

0	0	0	1	1	1
0	1	0	1	0	1
0	0	0	1	1	1

(a)

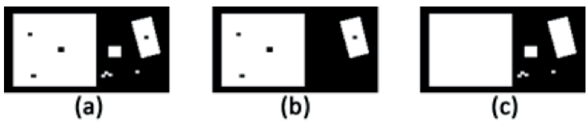
0	0	0	1	1	1
0	0	0	1	0	1
0	0	0	1	1	1

(b)

0	0	0	1	1	1
0	1	0	1	1	1
0	0	0	1	1	1

(c)

Figura 36 – Exemplo de Abertura e Fechamento por Área em uma Imagem Sintética. (a) Imagem de Entrada (b) Resultado da Abertura por Área (c) Resultado do Fechamento por Área

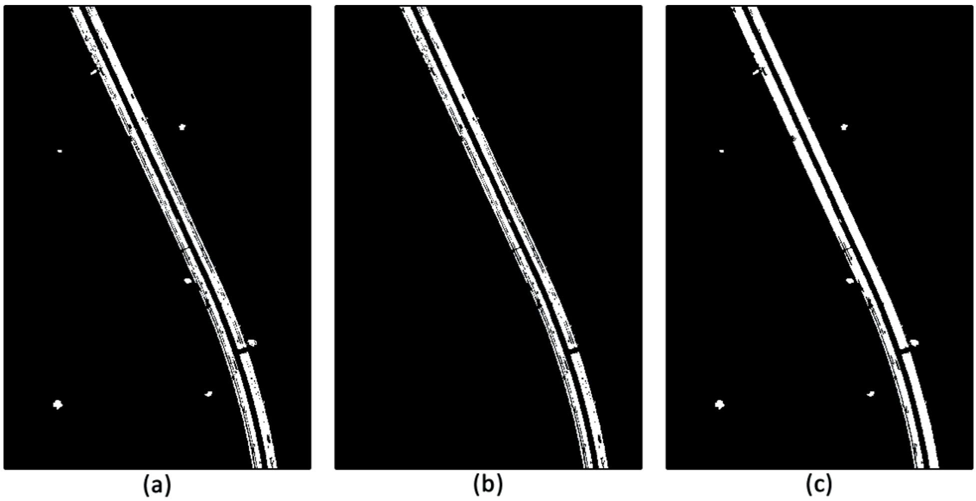


Ambas as funções possuem o mesmo comportamento computacionalmente, exceto pela tonalidade do alvo a ser retirado da imagem. No caso da Abertura por Área os pequenos alvos brancos são removidos, enquanto no Fechamento por Área são removidos os pequenos alvos pretos. Dessa forma, desenvolveu-se uma função, apresentada no Apêndice A pelo Quadro 62, que realiza ambas as funções, porém necessita de um parâmetro extra que identifica quais dos alvos serão removidos, brancos ou pretos. Essa função é posteriormente utilizada pelos operadores de Abertura e Fechamento por Área para proceder suas devidas aplicações. O Quadro 29 apresenta o modo de uso das funções de Abertura e Fechamento por Área enquanto a Figura 37 apresenta um resultado obtido pelo sistema para ambas as funções.

Quadro 29 – Como Aplicar as Funções de Abertura e Fechamento por Área.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/extraida.bmp");
cmStructureElement * se = new cmStructureElement(3,3,SE_BOX);
cmImage * imgOpen = cmFunctions::cmAreaOpen(imgOri,se,1000);
cmImage * imgClose = cmFunctions::cmAreaClose(imgOri,se,1000);
imgOpen->cmWriteImageToFile("C:/Users/Dissertacao/imagens/abertura_area.bmp");
imgClose->cmWriteImageToFile("C:/Users/Dissertacao/imagens/fechamento_area.bmp");
```

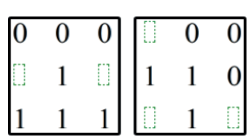
Figura 37 – Exemplo de Abertura e Fechamento por Área. (a) Imagem de Entrada (b) Resultado da Abertura por Área (c) Resultado do Fechamento por Área.



4.3.21 Afinamento

O operador de Afinamento tem por objetivo reduzir a espessura do alvo presente em uma imagem binária até que este possua apenas um *pixel* de largura. Para atingir este propósito, o operador de afinamento utiliza dois elementos estruturantes, apresentados na Figura 38, os quais são girados em todas as direções e então comparados com a vizinhança do *pixel* em análise. Todos os pontos onde tais elementos estruturantes coincidirem são eliminados do alvo de interesse. Este processo é repetido até o momento que não haja mais mudanças na imagem processada, o que indica que o alvo foi afinado. Vale ressaltar que na Figura 37 é possível observar algumas posições dos EE sem serem preenchidas, o que significa que esta posição do EE não importa na análise.

Figura 38 – Elementos Estruturantes Utilizados no Afinamento.

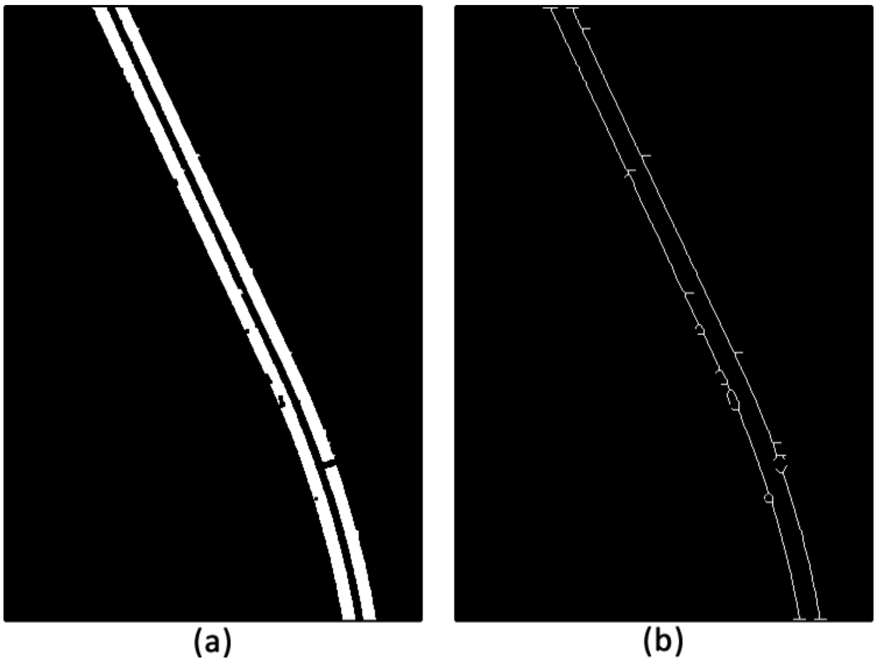


Para efeito de exemplificação, o algoritmo desenvolvido para esta função foi simplificado para processar apenas o primeiro elemento estruturante, apresentado na Figura 37, e é apresentado no Apêndice A pelo Quadro 63, enquanto o Quadro 30 demonstra o modo de uso dessa função. A Figura 39 ilustra um resultado obtido com a aplicação desta função em uma imagem binária.

Quadro 30 – Como Aplicar a Função de Afinamento.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/detectada.bmp");
cmImage * imgResult = cmFunctions::cmThinning(imgOri);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/afinamento.bmp");
```

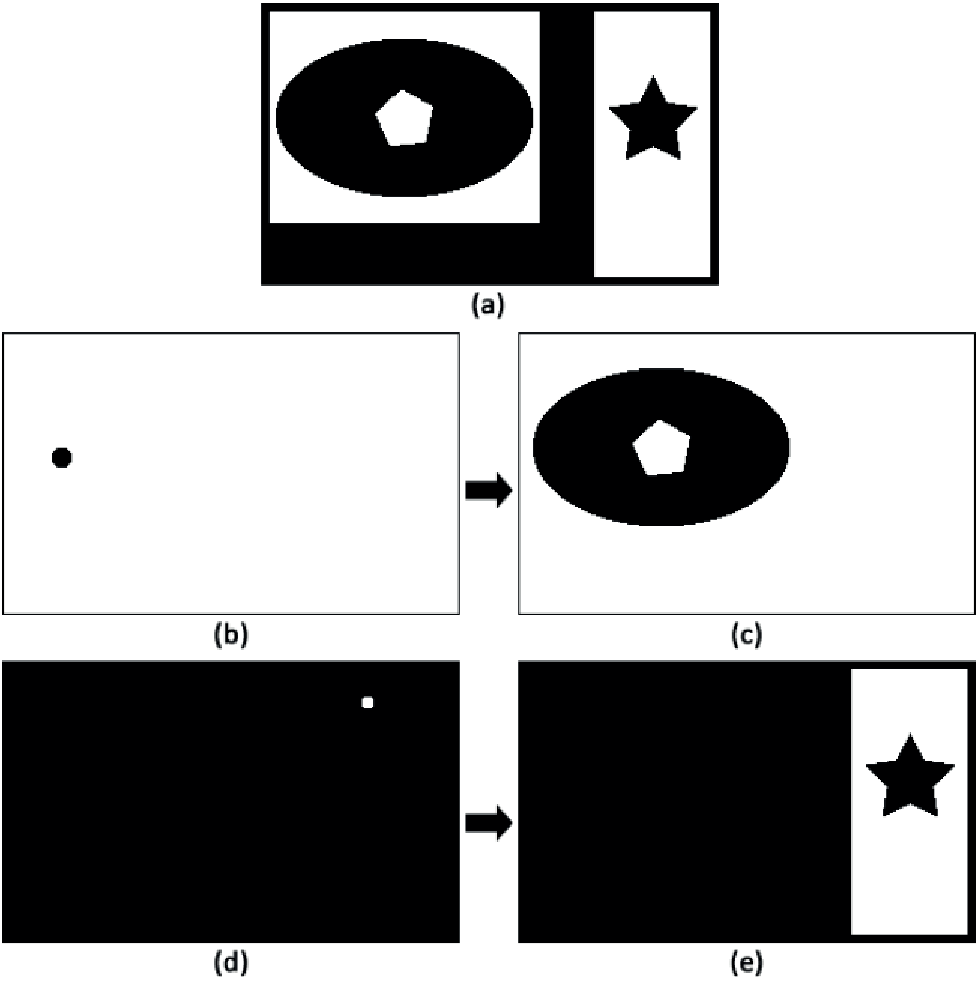
Figura 39 – Exemplo de Afinamento. (a) Imagem de Entrada (b) Resultado do Afinamento.



4.3.22 Erosão e Dilatação Condicional

Por se tratar de funções semelhantes, a Erosão e a Dilatação Condicionais serão apresentadas em conjunto neste tópico. Ambas as operações necessitam de duas imagens e um elemento estruturante como parâmetros de entrada. A primeira imagem necessária corresponde à imagem de referência, a qual deve ser uma imagem binária, enquanto que a segunda imagem será erodida, ou dilatada, de acordo com a imagem de referência, e, portanto, também deve ser binária. Este processo é repetido até que não haja mais modificações na imagem. Em outras palavras, um ponto presente na imagem de amostra é erodido, ou dilatada, até que o mesmo obtenha o formato do alvo correspondente na imagem de referência. Estas funções são comumente utilizadas para obter apena um dos alvos presentes na imagem de referência. A Figura 40 apresenta um exemplo sintético dos resultados obtidos com a aplicação da erosão e da dilatação condicional.

Figura 40 – Exemplo de Erosão e Dilatação Condicional. (a) Imagem de Referência (b) Imagem de Amostra (c) Resultado da Erosão Condicional (d) Imagem de Amostra (e) Resultado da Dilatação Condicional.

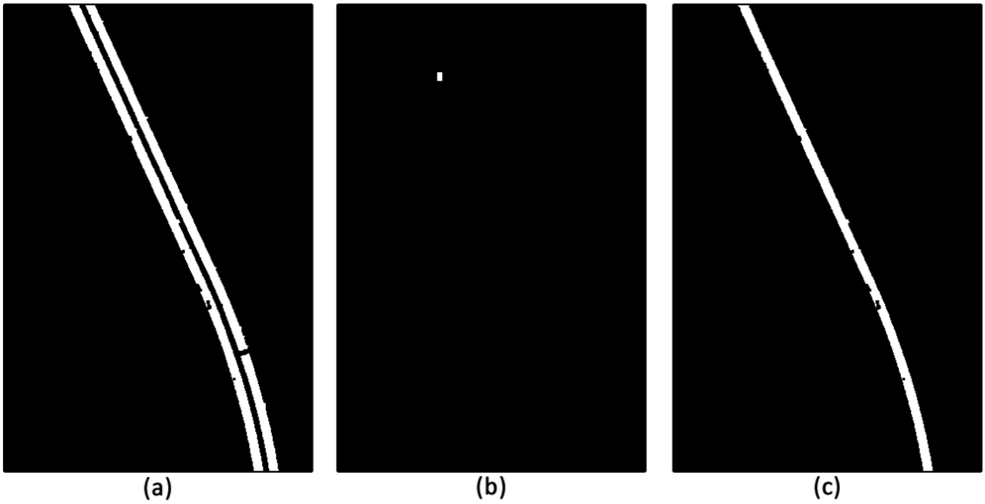


Os algoritmos desenvolvidos para estas funções estão apresentados no Quadro 64 (ver Apêndice A). Para exemplificação, o Quadro 31 apresenta o modo de uso da função de Dilatação Condicional, como o resultado obtido por esta operação é apresentado na Figura 41.

Quadro 31 – Como Aplicar a Função de Dilatação Condicional.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/detectada.bmp");
cmImage * imgAmostra = new cmImage("C:/Users/Dissertacao/imagens/dcond_amostra.bmp");
cmStructureElement * se = new cmStructureElement(3,3,SE_BOX);
cmImage * imgResult = cmFunctions::cmDilateCond(imgOri,imgAmostra,se);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/dilatacao_cond.bmp");
```

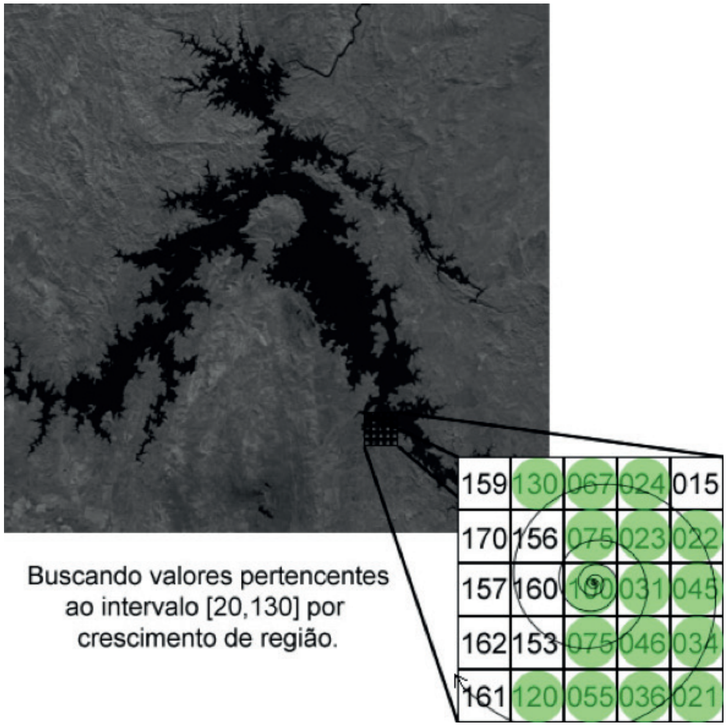
Figura 41 – Exemplo de Dilatação Condicional. (a) Imagem de Referência (b) Imagem de Amostra (c) Resultado da Dilatação Condicional.



4.3.23 Operador de Crescimento por Região.

O operador de Crescimento por Região tem um objetivo similar aos operadores de Erosão e Dilatação Condicionais, ou seja, utiliza uma imagem de amostra, onde o alvo presente nesta imagem irá se expandir, com base na imagem de referência, até obter apenas o alvo de interesse presente na imagem de referência. Contudo, enquanto que a imagem de referência na Erosão e Dilatação Condicionais deve ser uma imagem binária, no Crescimento por Região essa imagem pode ser uma imagem em tons de cinza. Esta diferença se deve a lógica de funcionamento do operador de Crescimento por Região, a qual é explicada a seguir. Para determinar se um *pixel* vizinho ao *pixel* em análise corresponde ao alvo de interesse, o algoritmo utiliza cálculos estatísticos, com base nos valores originais das amostras cedidas, para determinar um intervalo de valores que serão considerados como alvo de interesse. Em outras palavras, o algoritmo busca as posições dos *pixels* pertencentes a mostra e verifica os níveis de cinza dessas posições na imagem original. A partir desses valores é definido o intervalo de valores que devem ser considerados como pertencentes a feição de interesse. Definindo este intervalo, cada amostra é expandida para obter o alvo de interesse presente na imagem. A Figura 42 apresenta o funcionamento desse algoritmo.

Figura 42 – Funcionamento do Crescimento por Região.

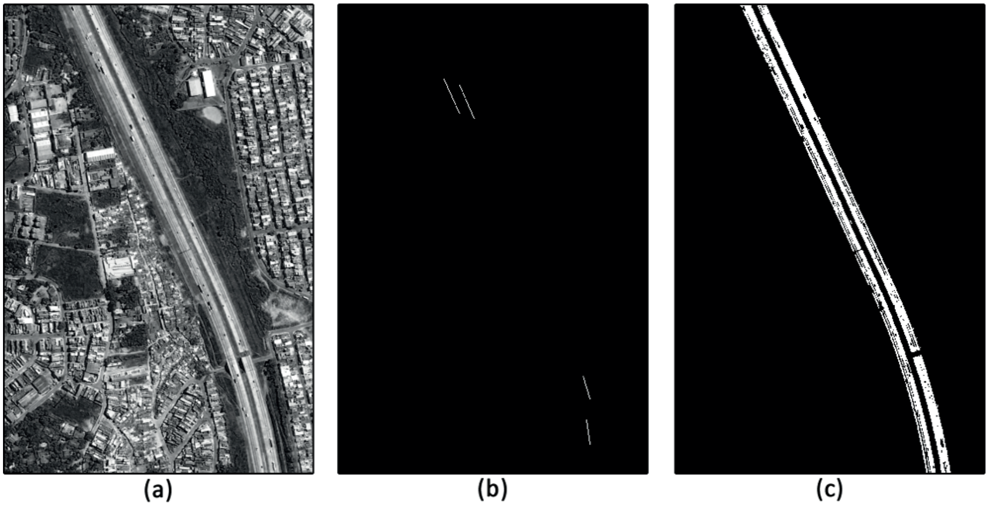


A função implementada para o operador de Crescimento por Região é apresentada no Apêndice A pelo Quadro 65, enquanto que o Quadro 32 apresenta o modo de uso dessa função e a Figura 43, o resultado obtido pela mesma.

Quadro 32 – Como Aplicar o Crescimento por Região.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/ori_b.bmp");
cmImage * imgAmostra = new cmImage("C:/Users/Dissertacao/imagens/amostra.bmp");
cmImage * imgResult = cmFunctions::cmGrowthRegion(imgOri,imgAmostra);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/cres_regiao.bmp");
```

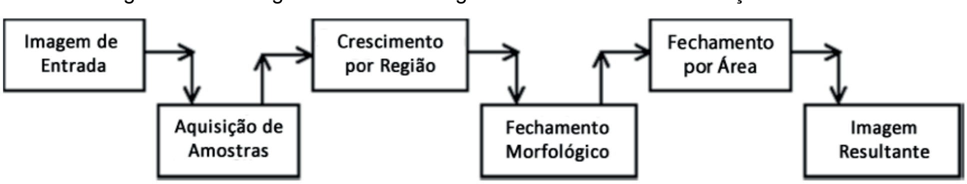
Figura 43 – Exemplo de Crescimento por Região. (a) Imagem Original (b) Imagem de Amostras (c) Resultado do Crescimento por Região.



4.3.24 Metodologia Semiautomática para Detecção de Feições Cartográficas.

Tendo como foco a detecção de feições cartográficas, principalmente rodovias e aeroportos, percebeu-se a necessidade de desenvolver uma rotina, ou algoritmo, para detectar tais feições a partir de determinada imagem. Esse algoritmo foi desenvolvido para facilitar a detecção de alvos cartográficos de interesse presentes em uma imagem. Ele foi considerado como semiautomático por ser baseado no operador de Crescimento por Região, e por este motivo, necessitar de informações do alvo de interesse, por meio da imagem de amostras, cedidas pelo usuário. Dessa forma, essa metodologia se torna dependente das amostras cedidas pelo usuário, uma vez que estas influenciam diretamente os cálculos do algoritmo. A metodologia desenvolvida consiste na aplicação de funções, previamente descritas, como apresentado pelo fluxograma da Figura 44.

Figura 44 – Fluxograma da Metodologia Semiautomática de Detecção de Alvos.

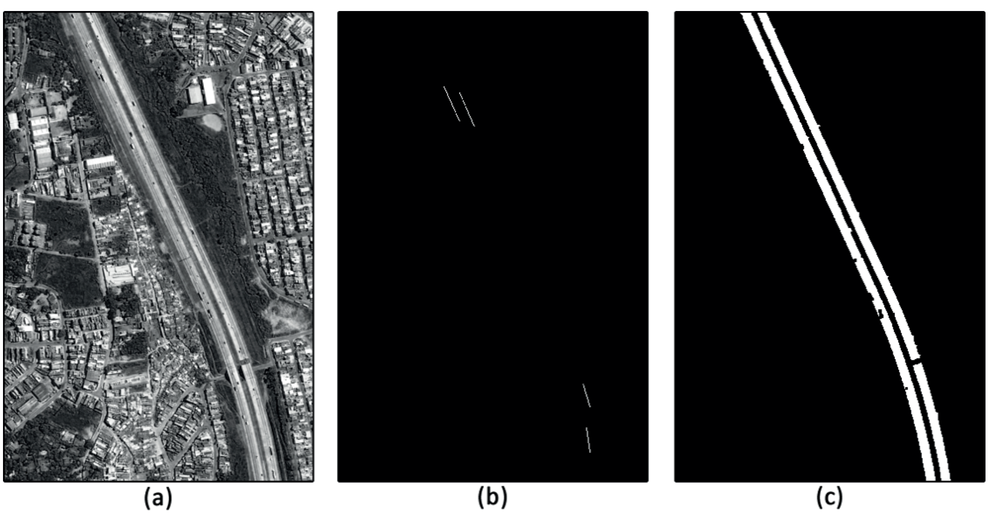


A rotina de detecção desenvolvida baseia-se no operador de crescimento por região para detectar o alvo de interesse. Contudo, o resultado desse operador pode ser melhorado pela funções de fechamento morfológico e fechamento por área posteriormente executadas. O fechamento morfológico é executado para conectar estruturas do alvo que foram erroneamente separadas durante o processo de crescimento por região, enquanto que o fechamento por área tem por objetivo incorporar pequenas áreas, presentes no interior da feição de interesse, ao alvo detectado (CARDIM et al., 2014). O algoritmo desenvolvido para essa função é apresentado no Apêndice A pelo Quadro 66. O Quadro 33 apresenta como utilizar essa função, enquanto que a Figura 45 apresenta o resultado obtido pela mesma.

Quadro 33 – Como Aplicar a Detecção de Feições.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/ori_b.bmp");
cmImage * imgAmostra = new cmImage("C:/Users/Dissertacao/imagens/amostra.bmp");
cmImage * imgResult = cmFunctions::cmFeatureDetection(imgOri,imgAmostra);
imgResult->cmWriteImageToFile("C:/Users/Dissertacao/imagens/detectada.bmp");
```

Figura 45 – Exemplo de Detecção de Feição. (a) Imagem Original (b) Imagem de Amostras (c) Resultado da Detecção de Feição.



4.4 CLASSE CMANALYSISVALUES

Pela importância no projeto de avaliar a qualidade do processo de extração, a análise estatística de extrações cartográficas foi implementada em uma classe independente. O objetivo dessa classe é fazer com que o usuário informe ao sistema, por meio do construtor da classe e utilizando os devidos parâmetros, as informações necessárias para realizar a análise estatística. O próprio construtor já realiza todas as operações necessárias. Sendo assim, todas as informações calculadas ficam armazenadas no objeto criado dessa classe e podem ser acessadas pelas funções presentes na classe. O Quadro 34 apresenta as funções disponíveis ao usuário nessa classe.

Quadro 34 – Funções da Classe cmAnalysisValues.

```
public:
    cmAnalysisValues(cmImage * imgOriginal, cmImage * imgReference, cmImage *
    imgExtracted, cmStructureElement * se);
    virtual ~cmAnalysisValues();
    int cmGetMatched();
    int cmGetUnmatchedRef();
    int cmGetUnmatchedExt();
    double cmGetCorrectness();
    double cmGetCompleteness();
    double cmGetQuality();
    cmImage * cmGetComparedImage();
    int cmGetMatchedExtWBuffer();
    int cmGetUnmatchedExtWBuffer();
    int cmGetMatchedRefWBuffer();
    int cmGetUnmatchedRefWBuffer();
    double cmGetCorrectnessWBuffer();
    double cmGetCompletenessWBuffer();
    double cmGetQualityWBuffer();
    double cmGetRedundancyWBuffer();
    double cmGetRMSWBuffer();
    cmImage * cmGetComparedReferenceImage();
    cmImage * cmGetComparedExtractedImage();
    void cmWriteToFile(char* fileName);
    double cmGetAverageRef();
    double cmGetStandardDeviationRef();
    int cmGetMedianRef();
    int cmGetModeRef();
    int cmGetMinRef();
    int cmGetMaxRef();
    double cmGetAverageExt();
    double cmGetStandardDeviationExt();
    int cmGetMedianExt();
    int cmGetModeExt();
    int cmGetMinExt();
    int cmGetMaxExt();
};
```

Para calcular as informações estatísticas da extração realizada, basta criar um objeto desta classe informando os devidos parâmetros necessários, que são: a imagem original; a imagem de referência; a imagem extraída; e o elemento estruturante, o qual será utilizado para criar a área de tolerância da análise. O Quadro 35 apresenta o modo de uso para obter todas as informações da análise estatística.

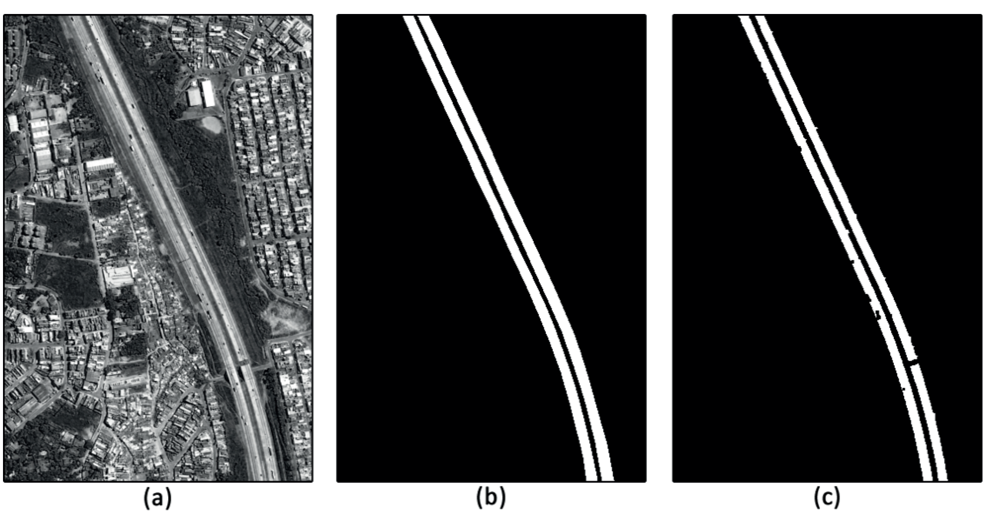
Quadro 35 – Como Aplicar a Análise Estatística do Resultado de uma Detecção.

```
cmImage * imgOri = new cmImage("C:/Users/Dissertacao/imagens/ori_b.bmp");
cmImage * imgExt = new cmImage("C:/Users/Dissertacao/imagens/detectada.bmp");
cmImage * imgRef = new cmImage("C:/Users/Dissertacao/imagens/referencia.bmp");
cmStructureElement * se = new cmStructureElement(3,3,SE_BOX);
cmAnalysisValues * analise = new cmAnalysisValues(imgOri,imgRef,imgExt,se);
analise->cmGetComparedImage()->cmWriteImageToFile(
"C:/Users/Dissertacao/imagens/comp_exata.bmp");
analise->cmGetComparedExtractedImage()->cmWriteImageToFile(
"C:/Users/Dissertacao/imagens/comp_ext.bmp");
analise->cmGetComparedReferenceImage()->cmWriteImageToFile(
"C:/Users/Dissertacao/imagens/comp_ref.bmp");
analise->cmWriteToFile("C:/Users/Dissertacao/imagens/analise.txt");
```

Calculados todos os valores, o usuário pode acessar todas as informações necessárias pelas funções apresentadas no Quadro 34 ou criar um arquivo texto com todas as informações usando a função *cmWriteToFile*, como exemplificado no Quadro 35.

Para mostrar os resultados obtidos por essa análise, a Figura 46 (a) é considerada como imagem original, tendo como alvo de interesse os dois trechos paralelos de uma rodovia; a Figura 46 (b) como a imagem de referência; e a Figura 46 (c) como a imagem extraída pelo método analisado.

Figura 46 – Exemplo da Análise Estatística. (a) Imagem Original (b) Imagem de Referência (c) Imagem Resultante da Detecção.



Considerando que a área de tolerância seja dada por um EE do tipo caixa de dimensões 3x3, o arquivo texto gerado com as informações da análise é apresentado no Quadro 36.

Quadro 36 – Informações da Análise Estatística.

```
The statistical values calculated during the analyze of the extraction method executed.

Traditional statistics about the reference image:
Average = 146.799
Mode = 137
Median = 145
Maximun = 241
Minimum = 13
Standard Deviation = 29.5089

Traditional statistics about the extracted image:
Average = 181.3
Mode = 178
Median = 180
Maximun = 254
Minimum = 24
Standard Deviation = 20.6889

Values calculated without the use of buffer:
Total Matched = 848
Unmatched Extraction = 1392
Unmatched Reference = 1339
Completeness = 0.387746
Correctness = 0.378571
Quality = 0.236938

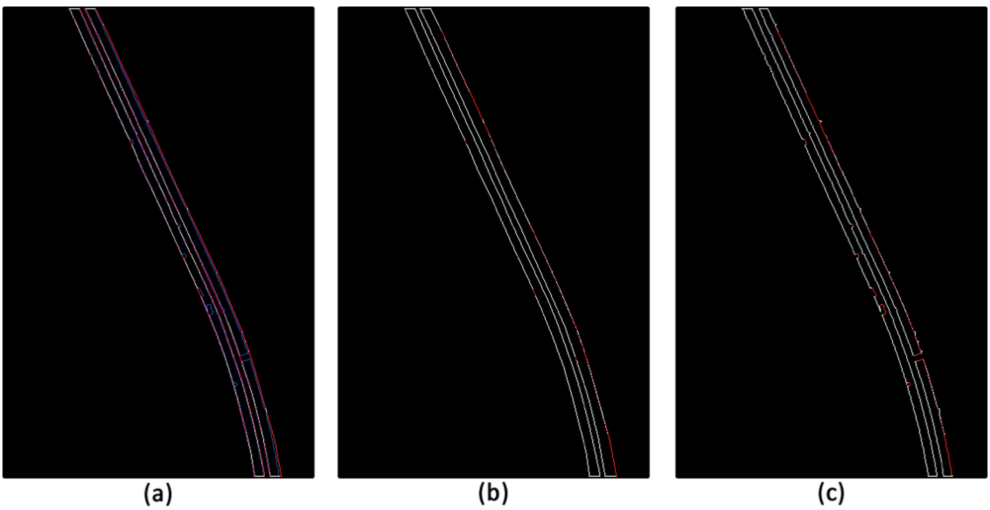
Values calculated with use of buffer:
Matched Extraction = 1918
Unmatched Extraction = 322
Matched Reference = 1904
Unmatched Reference = 283
Completeness = 0.870599
Correctness = 0.85625
Quality = 0.760206
Redundancy = 0.00729927
RMS = 1.1547
```

No Quadro 36 é possível observar quatro conjuntos de valores. O primeiro e o segundo grupo trazem informações estatísticas tradicionais, respectivamente, sobre as imagens de referência e detectada. Esses dois primeiros grupos apresentam valores como, média, mediana e desvio padrão, os quais são calculados utilizando-se os valores originais de brilho, presentes na imagem original, dos pontos considerados como pertencentes ao alvo de interesse pelas imagens de referência e detectada. Estas informações permitem comparar a imagem detectada com a imagem de referência com base nos valores de brilho da imagem original.

O terceiro conjunto de informações apresenta valores referentes à análise estatística da extração sem considerar área de tolerância, como sentado na Equação na Seção 2.4.1. Por outro lado, no quarto e último conjunto de informações apresentado pelo Quadro 36, encontram-se as informações estatísticas referentes à comparação que utiliza a área de tolerância definida pelo elemento estruturante informado. Essa ração é apresentada na Seção 2.4.1, com os cálculos definidos da Equação até a Equação (26).

Além das informações matemáticas, a análise estatística gera três imagens de comparação, seguindo finições apresentadas na Seção 2.4.1 e exemplificadas pelas Figuras 2 e 3. As imagens de comparação obtidas para o exemplo são apresentadas pela Figura 47, para a qual a Figura 47 (a) apresenta o resultado obtido durante a comparação exata; a Figura 47 (b), o resultado da correspondência sobre a imagem de referência; e a Figura 47 (c), o resultado da correspondência sobre a imagem extraída, ou detectada.

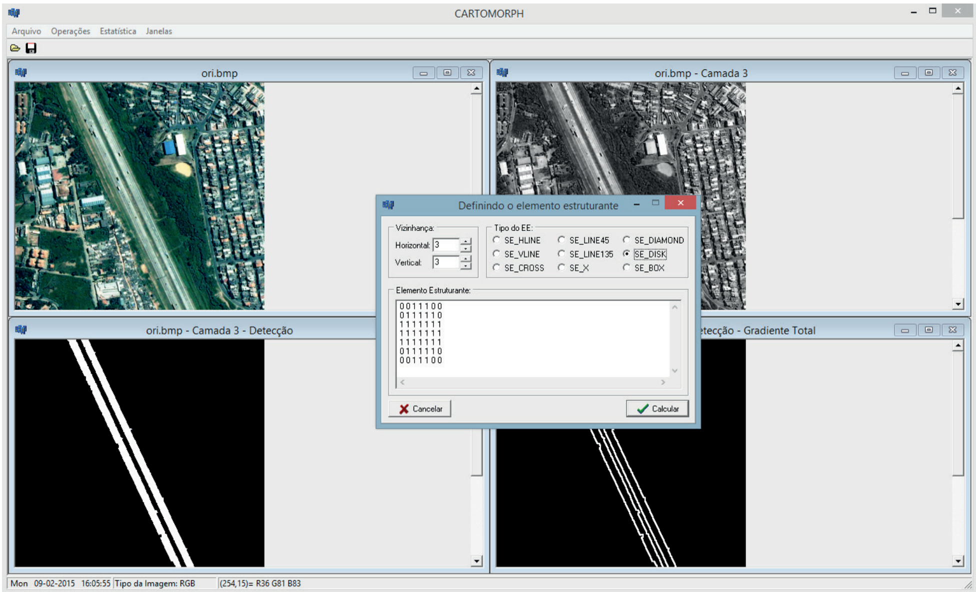
Figura 47 – Imagens Geradas na Análise Estatística. (a) Resultado da Comparação Exata (b) Resultado da Correspondência da Referência (c) Resultado da Correspondência da Extraída.



4.5 INTERFACE DO SISTEMA

Todas as funções apresentadas nas seções anteriores foram implementadas como uma biblioteca, possibilitando que o usuário as utilize em modo de programação, e permitindo o uso de outras técnicas e algoritmos necessários, inclusive de funções primitivas da linguagem de programação C/C++. Contudo, uma interface que utiliza essas funções foi implementada, possibilitando que o usuário sem conhecimento de programação possa realizar todas as operações desejadas e demonstradas anteriormente. A interface do sistema foi desenvolvida no ambiente C++ Builder, o qual permitiu uma total interação com os algoritmos desenvolvidos e possui recursos que facilitam a criação de interfaces. A interface desenvolvida é apresentada na Figura 48.

Figura 48 - Interface do Sistema CARTOMORPH.



4.6 DOCUMENTAÇÃO

Por se tratar do desenvolvimento de um sistema computacional, é necessário que o usuário tenha conhecimento do funcionamento, de como utilizar, como está implementado, dentre outras informações sobre o sistema. Sendo assim, é necessário a criação de uma documentação completa sobre o sistema desenvolvido. Neste projeto, a documentação foi criada de duas maneiras com o objetivo de facilitar o acesso às informações necessárias. Em um primeiro momento, toda função desenvolvida é comentada diretamente no código fonte, como pode ser observado nos algoritmos apresentados no Apêndice A, o que facilita o uso e a alteração de cada função diretamente no código fonte do sistema. Além disso, relatórios contendo informações conceituais; da implementação realizada e do modo de uso do sistema foram desenvolvidos e disponibilizados juntamente com o sistema. A documentação em forma de relatório foi desenvolvida por se tratar de uma documentação mais completa e mais detalhada do sistema do que apenas a documentação realizada em código fonte.

CONCLUSÕES E RECOMENDAÇÕES

Com o propósito de obter um sistema de processamento de imagens focado em detecção e/ou extração de feições cartográficas com base na morfologia matemática, uma biblioteca de funções foi desenvolvida em linguagem de programação C/C++, juntamente com uma interface que facilita a aplicação das operações por usuários sem conhecimento sobre programação. O sistema desenvolvido é de livre acesso, possibilitando uso, modificações e implementação de melhorias, o que permite que o sistema abranja um número cada vez maior de funcionalidades e aplicações.

A ideia de desenvolver um novo sistema foi considerada em razão da ausência de sistemas morfológicos de livre acesso aplicados à Cartografia. Como exemplo, os sistemas atuais são normalmente de domínio privado, o que impede a incorporação de novas funcionalidades, modificações e consequentemente adaptações à aplicação desejada. Além disso, o sistema desenvolvido possui foco diretamente no processamento de imagens para detecção e/ou extração de feições cartográficas de interesse e, portanto, possui funções próprias desse contexto, como uma metodologia semiautomática para a extração de feições e a análise estatística do resultado de extrações, as quais não foram encontradas em outros sistemas disponíveis. Sendo assim, o sistema é capaz de sanar as limitações encontradas no uso de outros sistemas, em relação ao contexto de sensoriamento remoto. Por esses motivos, esse projeto contribui positivamente para a área de Cartografia.

Os resultados obtidos por todas as operações implementadas foram analisados para verificar a correta implementação das operações e consequentemente do correto processamento das imagens. Desta maneira, o sistema mostrou-se eficiente na execução de todas as operações implementadas. Além disso, houve a preocupação de realizar todas as operações com o menor custo computacional possível, diminuindo, assim, o tempo de execução e espera do usuário. Para tanto, cálculos desnecessários e/ou repetitivos foram evitados, assim como o uso indevido de memória para o processamento.

Com a implementação da metodologia semiautomática de detecção/extração de feições cartográficas, mostra-se a eficácia do sistema desenvolvido que possui por foco possibilitar a implementação de novas metodologias de detecção e/ou extração de feições cartográficas baseadas na teoria da morfologia matemática. Além disso, o sistema permite que os resultados obtidos por essas metodologias sejam avaliados estatisticamente, contribuindo diretamente no desenvolvimento e testes dessas metodologias.

Embora contenha funcionalidades suficientes para realizar detecções e/ou extrações de feições cartográficas de interesse, sugere-se que outras técnicas e operadores de PDI sejam estudadas quanto a sua importância e consequentemente incluídas no sistema desenvolvido.

REFERÊNCIAS

BACHER, U.; MAYER, H. Automatic Road Extraction from Multispectral High Resolution Satellite Images. (U. Stilla, F. Rottensteiner, S. Hinz, Eds.) In: Proceedings of the ISPRS Workshop CMRT 2005, Vienna. **Anais**... Vienna: ISPRS, 2005. Disponível em: <http://www.isprs.org/proceedings/XXXVI/3-W24/papers/CMRT05_Bacher_Mayer.pdf>.

BANON, G. J. F. Minicurso. In: X Congresso Nacional de Matemática Aplicada e Computacional, Gramado. **Anais**... Gramado: 1987.

BANON, G. J. F.; BARRERA, J. **Bases da morfologia matemática para análise de imagens binárias**. 2. ed. São José dos Campos: INPE, 1998.

BAXES, G. A. **Digital image processing: principles and applications**. Michigan: New York, 1994.

BELLENS, R.; GAUTAMA, S.; MARTINEZ-FONTE, L.; PHILIPS, W.; CHAN, J. C.; CANTERS, F. Improved Classification of VHR Images of Urban Areas Using Directional Morphological Profiles. **IEEE Transactions on Geoscience and Remote Sensing**, v. 46, n. 10, p. 2803–2813, 2008. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4637924>.

CARDIM, G. P.; DA SILVA, E. A.; DIAS, M. A.; BRAVO, I. Semiautomatic Methodology for Cartographic Features Extraction Using High-Resolution Remote Sensing Images. In: Proceedings of the XVI Simposio Internacional SELPER, Medellin. **Anais**... Medellin: SELPER, 2014. Disponível em: <http://www.selper.org.co/capitulo_colombia/papers/Fotogrametria-PDI-Fusion-de-datos/FP22-Semiautomatic-methodology-Features-extraction.pdf>.

CARDIM, G. P.; SILVA, E. A. da. Análise da Qualidade de Processos Automáticos de Extração de Feições Cartográficas. **Omnia Exatas**, v. 4, n. 2, p. 7–18, 2011. Disponível em: <http://www.fai.com.br/portal/ojs/index.php/omniaexatas/article/view/99/pdf>.

CARDIM, G. P.; SILVA, E. A. da. Development of an Algorithm to Analyze Cartographic Features Extraction Methods. In: Anais do 35 International Symposium on Remote Sensing of Environment, **Anais**...2013.

CROSTA, A. P. **Processamento digital de imagens de sensoriamento remoto**. Campinas: UNICAMP/Instituto de Geociências, 1999.

DAL POZ, A. P. **Metodologia semi-automática para extração de rodovias em imagens digitais usando programação dinâmica, análise de bordas de rodovia e teste ativo**. 2005. Universidade Estadual Paulista "Júlio de Mesquita Filho" Faculdade de Ciências e Tecnologia (FCT/UNESP), 2005.

DAL POZ, A. P.; DO VALE, G. M.; ZANIN, R. B. Automatic extraction of road seeds from high-resolution aerial images. **Anais da Academia Brasileira de Ciências**, v. 77, n. 3, p. 509–520, 2005.

DAL POZ, A. P.; ZANIN, R. B.; DO VALE, G. M. Extração Automática de Feições Rodoviárias em Imagens Digitais. **Sba Controle & Automação**, v. 18, n. 1, p. 44–54, 2007. Disponível em: <http://www.scielo.br/pdf/ca/v18n1/a04v18n1.pdf>.

DO VALE, G. M.; ZANIN, R. B.; DAL POZ, A. P. Limiarização Contextual Automática de Imagens Coloridas: Aplicação na extração de sementes de rodovia. **Boletim de Ciências Geodésicas**, v. 14, n. 1, p. 72–93, 2008.

DOUGHERTY, E. R.; LOTUFO, R. A. **Hands-on morphological image processing**. [s.l.] SPIE Press, 2003.

DURAND, F.; DORSEY, J. Fast Bilateral Filtering for the Display of High-Dynamic-Range Images. **ACM Transactions on Graphics**, v. 21, n. 3, p. 257–266, 2002. Disponível em: <http://dx.doi.org/10.1145/566570.566574>.

FACON, J. **Morfologia matemática: teoria e exemplos**. XII ed.Curitiba: Editora Universitária Champagnat da Pontifícia Universidade Católica do Paraná, 1996.

GALLIS, R. B. de A. **Extração semi-automática da malha viária em imagens aéreas digitais de áreas rurais utilizando otimização por programação dinâmica no espaço objeto**. 2006. Universidade Estadual Paulista "Júlio de Mesquita Filho" Faculdade de Ciências e Tecnologia (FCT/UNESP), 2006. Disponível em: <http://www2.fct.unesp.br/pos/cartografia/docs/teses/t_gallis_rba.pdf>.

GÉRAUD, T.; MOURET, J.-B. Fast Road Network Extraction in Satellite Images Using Mathematical Morphology and Markov Random Fields. **EURASIP Journal on Advances in Signal Processing**, v. 2004, n. 16, p. 2503–2514, 2004. Disponível em: <http://asp.eurasipjournals.com/content/2004/16/473593>.

GOMES, J.; VELHO, L. **Computação gráfica: imagem**. Rio de Janeiro: Impa/sbm, 1994.

GONZALEZ, R. C.; WOODS, R. E. **Processamento digital de imagens**. 3. ed. São Paulo: Pearson Prentice Hall, 2010.

GONZALEZ, R. C.; WOODS, R. E.; EDDINS, S. L. **Digital image processing: using matlab**. [s.l.] Pearson Prentice Hall, 2004.

GOUTSIAS, J.; HEIJMANS, H. J. A. M. **Mathematical morphology**. Amsterdam: IOS Press, 2000.

HINZ, S.; BAUMGARTNER, A. Road Extraction in Urban Areas Supported by Context Objects. **International Archives of Photogrammetry and Remote Sensing**, v. XXXIII, p. 405–412, 2000. Disponível em: <http://www.isprs.org/proceedings/xxxiii/congress/part3/405_XXXIII-part3.pdf>.

INTERNATIONAL TELECOMMUNICATION UNION. **Recommendation ITU-R BT.601-4 - Enconding Parameters Of Digital Television For Studios**. [s.l: s.n.]. Disponível em: <http://www.itu.int/rec/R-REC-BT.601-4-199407-S/en>.

ISHIKAWA, A. S. **Detecção de rodovias em imagens digitais de alta resolução com o uso da teoria de morfologia matemática**. 2008. 9 Universidade Estadual Paulista "Júlio de Mesquita Filho" Faculdade de Ciências e Tecnologia (FCT/UNESP), 2008. Disponível em: <http://www2.fct.unesp.br/pos/cartografia/docs/teses/d_ishikawa_as.pdf>.

ISHIKAWA, A. S.; SILVA, E. A. da; NÓBREGA, R. A. de A. Extração de Rodovias em Imagens Digitais de Alta Resolução com o Uso da Teoria de Morfologia Matemática. **Revista Brasileira de Cartografia**, p. 131–140, 2010. Disponível em: <http://www.lsie.unb.br/rbc/index.php/rbc/article/view/365/357>.

LAMPINEN, J.; LAAKSONEN, J.; OJA, E. Pattern Recognition. In: LEONDES, C. T. (Ed.). **Image Processing and Pattern Recognition**. [s.l.] Elsevier Science, 1998. p. 386.

LEE, J. S. J.; HARALICK, R. M.; SHAPIRO, L. G. Morphologic edge detection. **IEEE Journal on Robotics and Automation**, v. 3, n. 2, p. 142–156, abr. 1987. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1087088>.

LILLESAND, T. M.; KIEFER, R. W.; CHIPMAN, J. W. **Remote sensing and image interpretation**. 5. ed. New jersey: John Wiley & Sons, 2007.

MASCARENHAS, N. D.; SILVA, E. A. da. Analysis of the Performance of Morphological Edge Detectors with Respect to Edge Orientation and Displacement. (C. M. D. S. Freitas, R. M. Persiano, Eds.) In: Anais do Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, 1, Gramado. **Anais**... Gramado: Sociedade Brasileira de Computação, 1990. Disponível em: <http://urlib.net/sid.inpe.br/sibgrapi/2012/09.19.17.03>.

MATHERON, G. **Random sets and integral geometry**. New York: John Wiley, 1974.

MCANDREW, A. **Introduction to digital image processing with matlab**. Victoria: Thomson, 2004.

MOHAMMADZADEH, A.; TAVAKOLI, A.; ZOEJ, M. J. V. Road Extraction Based on Fuzzy Logic Pan-Sharpened IKONOS Images. **The Photogrammetric Record**, v. 21, n. 113, p. 44–60, 2006. Disponível em: <http://onlinelibrary.wiley.com/doi/10.1111/j.1477-9730.2006.00353.x/pdf>.

NÓBREGA, R. A. de A. **Detecção de malha viária na periferia urbana de são paulo utilizando imagens orbitais de alta resolução espacial e classificação orientada a objetos**. 2007. Escola Politécnica da Universidade de São Paulo, 2007.

PARKER, J. R. **Algorithms for image processing and computer vision**. 2. ed. Indianapolis: John Wiley & Sons, 2010.

PEDRINI, H.; SCHWARTZ, W. R. **Análise de imagens digitais: princípios, algoritmos e aplicações**. São Paulo: Thomson Learning, 2008.

PÉTERI, R.; CELLE, J.; RANCHIN, T. Detection and extraction of road networks from high resolution satellite images. In: IEEE International Conference on Image Processing, Barcelona. **Anais**... Barcelona: ICIP2003 Proceedings, 2003. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp= &arnumber=1246958&queryText%3DDetection+and+extraction+of+road+networks+from+high+resolution+satellite+images>.

RODRIGUES, T. G.; SILVA, E. A. da; LEONARDI, F. O uso de morfologia matemática na detecção de pistas em autódromo. **Revista Brasileira de Cartografia2**, v. 2, p. 337–343, 2010.

SANTOS, F. P.; SILVA, E. A. da; NÓBREGA, R. A. de A. Proposição de Rotina Morfológica para Detecção de Malha Viária em Imagens Orbitais. **Science & Engineering Journal**, v. 19, n. 1, p. 01–06, 2010.

SERRA, J. P.; CRESSIE, N. A. C. **Image analysis and mathematical morphology: vol. 1**. London: Academic Press, 1982.

SILVA, E. A. da; CARDIM, G. P. Applying Digital Image Processing to Evaluate a Extraction Method of Cartographic Features in Digital Images. **Journal of Earth Science and Engineering**, v. 2, n. 4, p. 241–246, 2012. Disponível em: <http://davidpublishing.org/show.html?5711>.

SILVA, E. A. da; CARDIM, G. P.; BEST, R. De. Semiautomatic Algorithm to Extraction of Cartographic Features in Digital Images. **Journal of Communication and Computer**, v. 9, p. 1247–1251, 2012.

SILVA, M. L. da; CARRARD, M. C. C.; D'ORNELLAS, M. C. MorphoLib : Uma Biblioteca Genérica para o Processamento Morfológico de Imagens. In: I Workshop de Computação da Região Sul, Florianópolis. **Anais**... Florianópolis: 2004. Disponível em: <http://inf.unisul.br/~ines/workcomp/cd/pdfs/2216.pdf>.

SOILLE, P. **Morphological image analysis: principles and applications**. 2. ed. Berlin: Springer, 2003.

STATELLA, T. **Detecção automática de rastros de dust devils na superfície de marte**. 2012. Universidade Estadual Paulista "Júlio de Mesquita Filho" Faculdade de Ciências e Tecnologia (FCT/UNESP), 2012. Disponível em: <http://www2.fct.unesp.br/pos/cartografia/docs/teses/t_statella_t.pdf>.

STATELLA, T.; SILVA, E. A. da. Morfologia matemática aplicada à detecção de sombras e nuvens em imagens de alta resolução. **Boletim de Ciências Geodésicas**, v. 14, n. 2, p. 256–271, 2008. Disponível em: <http://thiagostatella.weebly.com/uploads/7/4/6/3/7463160/bcg2008.pdf>.

WIEDEMANN, C. External Evaluation of Road Networks. **ISPRS Archives**, v. XXXIV, n. 3, p. 93–98, 2003. Disponível em: <http://www.isprs.org/proceedings/xxxiv/3-w8/papers/pia03_s4p2.pdf>.

WIEDEMANN, C.; HEIPKE, C.; MAYER, H.; HINZ, S. Automatic Extraction and Evaluation of Road Networks From MOMS-2P Imagery. **International Archives of Photogrammetry and Remote Sensing**, p. 95–100, 1998. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.5837>.

YAN, D.; ZHAO, Z. Road Detection From Quickbird Fused Image Using IHS Transform and Morphology. In: International Geoscience and Remote Sensing Symposium, Toulose. **Anais**... Toulose: Proceedings, IGARSS 2003, 2003.

Quadro 37 – Algoritmo da Função Construtora para Abrir Imagens – cmImage.

```
/**
 * The class constructor. Create an image using an image file.
 * @param fileName The name of the image file to Open. It works just with the
 extensions '.bmp'.
 */
cmImage::cmImage(char * fileName) {
    //using the EASYBMP library to open a bmp file
    BMP Input;
    Input.ReadFromFile(fileName);

    int i, j;

    //checking if the image is not a grayscale
    if (Input.TellBitDepth() > 8) {
        //initializing the image as a RGB Color image
        cmInitImage(Input.TellWidth(), Input.TellHeight(), RGBCOLOR);

        //copying all the open image to the cmImage
        for (i = 0; i < Input.TellHeight(); i++) {
            for (j = 0; j < Input.TellWidth(); j++) {
                //The pixel of coordinates (i,j) on cartomorph library is the
                pixel of coordinates (j,i) on EASYBMP library
                cmSetPixel(i, j, (int) Input(j, i)->Red, (int) Input(j, i)->Green,
                (int) Input(j, i)->Blue);
            }
        }
        return;
    }

    if (Input.TellBitDepth() == 1)
        //creating a image in binarymode
        cmInitImage(Input.TellWidth(), Input.TellHeight(), BINARY);
    else
        //creating a image in grayscale mode
        cmInitImage(Input.TellWidth(), Input.TellHeight(), GRAYSCALE);

    //copying all the open image to the cmImage
    for (i = 0; i < Input.TellHeight(); i++) {
        for (j = 0; j < Input.TellWidth(); j++) {
            cmSetRColor(i, j, (int) Input(j, i)->Red);
        }
    }

}
```

Quadro 38 – Algoritmo da Função para Salvar a Imagem no Disco – cmWriteImageToFile.

```
/**
 * Write an image file of the image.
 * @param fileName The file name to be created.
 */
void cmImage::cmWriteImageToFile(char * fileName) {

    if (this == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmWriteImageToFile:
        Cannot write the file. The image is NULL.";
        return;
    }
    if (fileName == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmWriteImageToFile:
        Cannot write the file with a NULL fileName.";
        return;
    }

    BMP Output;

    Output.SetSize(cmGetWidth(), cmGetHeight());

    int i, j;

    if (cmIsRGB()) {

        //copying the image to the EASYBMP library
        for (i = 0; i < cmGetHeight(); i++) {
            for (j = 0; j < cmGetWidth(); j++) {
                Output(j, i)->Red = cmGetPixel(i, j, RED);
                Output(j, i)->Green = cmGetPixel(i, j, GREEN);
                Output(j, i)->Blue = cmGetPixel(i, j, BLUE);
            }
        }
        //writing the image to a file
        Output.WriteToFile(fileName);
        return;
    }

    //writing a grayscale image
    int aux;
    //copying the image to the EASYBMP library
    for (i = 0; i < cmGetHeight(); i++) {
        for (j = 0; j < cmGetWidth(); j++) {
            aux = cmGetRColor(i, j);
            Output(j, i)->Red = aux;
            Output(j, i)->Green = aux;
            Output(j, i)->Blue = aux;
        }
    }
    //creating a grayscale color table for the image
    if (cmIsGrayScale()) {
        Output.SetBitDepth(8);
        CreateGrayscaleColorTable(Output);
    } else Output.SetBitDepth(1);


    //writing the image to a file
    Output.WriteToFile(fileName);

}
```

Quadro 39 – Algoritmo da Função de Binarização – cmGrayToBinary.

```
/**
 * Transform the grayscale image to a binary image.
 * @param threshold The value of decision if the pixel is black or white.
 * @return The image created just with BLACK and WHITE colors.
 */
cmImage * cmImage::cmGrayToBinary(int threshold) {

    if (this == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGrayToBinary: Cannot
        transform a NULL image to binary.";
        return NULL;
    }

    if (this->cmIsGrayScale()) {
        int i, j;
        //creating the Binary image result
        cmImage * imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(),
        BINARY);

        //scanning the image
        for (i = 0; i < this->cmGetHeight(); i++) {
            for (j = 0; j < this->cmGetWidth(); j++) {
                if (this->cmGetRColor(i, j) >= threshold)
                    imgResult->cmSetRColor(i, j, WHITE);
                else
                    imgResult->cmSetRColor(i, j, BLACK);
            }
        }
        return imgResult;
    }
    cout << endl << "CARTOMORPH ERROR in the function cmGrayToBinary: The image is
    not a grayscale image.";
    return NULL;
}
```

Quadro 40 – Algoritmo da Função de Inversão – cmInvertImage.

```
/**
 * Invert the colors of an image.
 * @return The new image with the values inverted.
 */
cmImage * cmImage::cmInvertImage() {

    if (this == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmInvertImage: Cannot invert the colors of a NULL image.";
        return NULL;
    }

    int i, j;
    if (cmIsRGB()) {

        //creating the RGB Color image result
        cmImage * imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(), RGBCOLOR);

        for (i = 0; i < this->cmGetHeight(); i++) {
            for (j = 0; j < this->cmGetWidth(); j++) {
                imgResult->cmSetPixel(i, j,
                        255 - this->cmGetRColor(i, j),
                        255 - this->cmGetGColor(i, j),
                        255 - this->cmGetBColor(i, j)
                        );
            }
        }
        return imgResult;
    }
    //inverting a grayscale image

    //creating the grayscale image result
    cmImage * imgResult;
    if (cmIsGrayScale())
        imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(), GRAYSCALE);
    else
        imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(), BINARY);

    for (i = 0; i < this->cmGetHeight(); i++) {
        for (j = 0; j < this->cmGetWidth(); j++) {
            imgResult->cmSetRColor(i, j, 255 - this->cmGetRColor(i, j));
        }
    }

    return imgResult;
}
```

Quadro 41 – Algoritmo da Função de Conversão de RGB para tons de cinza – cmRGBToGray.

```
/**
 * Transform the image to grayscale.
 * @return Return the image created in grayscale.
 */
cmImage * cmImage::cmRGBToGray() {

    if (this == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmRGBToGray: Cannot transform a NULL image to GRAYSCALE.";
        return NULL;
    }

    int i, j, aux;

    //checking if the image is a RGB Color image
    if (cmIsRGB() || cmIsBinary()) {

        //creating the grayscale image result
        cmImage * imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(), GRAYSCALE);

        //scanning the image
        for (i = 0; i < this->cmGetHeight(); i++) {
            for (j = 0; j < this->cmGetWidth(); j++) {
                aux = (int) (0.299 * this->cmGetRColor(i, j)
                        + 0.587 * this->cmGetGColor(i, j)
                        + 0.114 * this->cmGetBColor(i, j));
                imgResult->cmSetRColor(i, j, aux);
            }
        }
        return imgResult;
    }
    cout << endl << "CARTOMORPH ERROR in the function cmRGBToGray: Cannot transform a GRAYSCALE image to GRAYSCALE.";
    return NULL;
}
```

Quadro 42 – Algoritmo da Função de Equalização de Histograma - cmEqualizeHistogram

```
/**
 * Perform a transformation for equalize the image histogram.
 * @return The result image.
 */
cmImage * cmImage::cmEqualizeHistogram() {

    if (this == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmEqualizeHistogram: The
        image is NULL.";
        return NULL;
    }
    if (cmIsBinary()) {
        cout << endl << "CARTOMORPH ERROR in the function cmEqualizeHistogram:
        This transformation cannot be performed using a binary image.";
        return NULL;
    }
    int * histogram = this->cmGetHistogram();
    int * histogramRelative = this->cmGetHistogramRelative(histogram);

    int i, j, k, b, aux = 0;

    if (this->cmIsRGB()) {
        b = 3;
    } else {
        b = 1;
    }

    cmImage * imgResult = new cmImage(this->cmGetWidth(), this->cmGetHeight(), this
    ->cmGetImageType());

    for (k = 0; k < b; k++) {
        for (i = 0; i<this->cmGetHeight(); i++) {
            for (j = 0; j<this->cmGetWidth(); j++) {
                aux = floor((double)((255 * (*(histogramRelative + k * 256 + this->
                cmGetPixel(i, j, k)))) / (this->cmGetWidth() * this->cmGetHeight
                ())));
                aux = MAX(0, aux);
                imgResult->cmSetPixel(i, j, k, aux);
            }
        }
    }
    return imgResult;
}
```

Quadro 43 – Algoritmo da Função de Suavização por Filtro da Média – cmFilterAVG.

```cpp
/**
 * Create an image blurred with the average value calculated using the structure element.
 * @param img The image to be blurred.
 * @param se The structure element.
 * @return The blurred image create.
 */
cmImage * cmFunctions::cmFilterAVG(cmImage * img, cmStructureElement * se) {

    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterAVG: Cannot perform the FilterAVG
        operation using a NULL image.";
        return NULL;
    }
    if (img->cmIsBinary()) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterAVG: Cannot perform the FilterAVG
        operation using a binary image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterAVG: Cannot perform the FilterAVG
        operation using a NULL Structure Element.";
        return NULL;
    }

    int i, j, x, y, sum, seDim, seDimSub, dimw, dimh;
    int auxIDimh, auxIDimwx, auxJDimw;

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), img->cmGetImageType());

    //discovering the neighborhood size
    dimh = se->cmGetHeight() / 2;
    dimw = se->cmGetWidth() / 2;

    seDim = se->cmGetHeight() * se->cmGetWidth();

    //scanning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dimh;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dimw;
            //the first minimun value is the maximum value possible
            sum = 0;
            seDimSub = 0;
            //scanning the structure element
            for (x = 0; x < se->cmGetHeight(); x++) {
                //geting a coordinate to be accessed on the image
                auxIDimwx = auxIDimh + x;
                //if the coordinate is smaller than 0 or bigger than the image dimension
                //the coordinate is outside of the image and cannot do the calculus, so
                //the algorithm goes to the next coordinate possible
                if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) {
                    seDimSub += se->cmGetWidth();
                    continue;
                }
                for (y = 0; y < se->cmGetWidth(); y++) {
                    //the same problem with the coordinates outside of the image
                    if ((auxJDimw + y < 0) || (auxJDimw + y) >= img->cmGetWidth()) {
                        seDimSub++;
                        continue;
                    }
                    if (se->cmGetValue(x, y) == 1) {
                        sum += img->cmGetRColor(auxIDimwx, auxJDimw + y);
                    } else seDimSub++;
                }
            }
            //allocatig the minimum value to the image coordinates
            imgResult->cmSetRColor(i, j, sum / (seDim - seDimSub));
        }
    }
    //returning the image result
    return imgResult;

}
```

Quadro 44 - Algoritmo da Função de Suavização por Filtro da Mediana– cmFilterMedian

```
/**
 * Create an image blurred with the median value calculated using the structure element.
 * @param img The image to be blurred.
 * @param se The structure element.
 * @return The blurred image create.
 */
cmImage * cmFunctions::cmFilterMedian(cmImage * img, cmStructureElement * se) {

    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterMedian: Cannot execute the FilterMedian operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterMedian: Cannot execute the FilterMedian operation using a NULL Structure Element.";
        return NULL;
    }

    int i, j, x, y, sum, seDim, seDimSub, dimw, dimh;
    int auxIDimh, auxIDimwx, auxJDimw;

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), img->cmGetImageType());

    //discovering the neighborhood size
    dimh = se->cmGetHeight() / 2;
    dimw = se->cmGetWidth() / 2;

    seDim = se->cmGetHeight() * se->cmGetWidth();

    //scanning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dimh;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dimw;
            seDimSub = 0;
            //creating an auxiliar list
            list<int> listAux;
            //scanning the structure element
            for (x = 0; x < se->cmGetHeight(); x++) {
                //geting a coordinate to be accessed on the image
                auxIDimwx = auxIDimh + x;
                //if the coordinate is smaller than 0 or bigger than the image dimension
                //the coordinate is outside of the image and cannot do the calculus, so
                //the algorithm goes to the next coordinate possible
                if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) {
                    seDimSub += se->cmGetWidth();
                    continue;
                }
                for (y = 0; y < se->cmGetWidth(); y++) {
                    //the same problem with the coordinates outside of the image
                    if ((auxJDimw + y < 0) || (auxJDimw + y) >= img->cmGetWidth()) {
                        seDimSub++;
                        continue;
                    }
                    if (se->cmGetValue(x, y) == 1) {
                        //inserting the value in the list end
                        listAux.push_back(img->cmGetRColor(auxIDimwx, auxJDimw + y));
                    } else seDimSub++;
                }
            }
            //sorting the values
            listAux.sort();
            int listSize = listAux.size();
            //going to list central position
            for (int p = 0; p < (listSize / 2); p++)
                listAux.pop_back();
            //allocatig the median value to the image coordinates
            imgResult->cmSetRColor(i, j, listAux.back());
        }
    }
    //returning the image result
    return imgResult;

}
```

Quadro 45 – Algoritmo da Função do Filtro Laplaciano – cmFilterEdgesLaplace.

```
/**
 * Execute the Laplace filter in an image and return the image result.
 * @param img The image to calculate the laplace edge detector.
 * @return The image result.
 */
cmImage * cmFunctions::cmFilterEdgesLaplace(cmImage * img) {

    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmFilterEdgesLaplace:
        Cannot perform the Laplace filter in a NULL image.";
        return NULL;
    }

    int seValues[] = {0, -1, 0, -1, 4, -1, 0, -1, 0};
    int dim = 3, dim2 = dim / 2;

    cmStructureElement * se = new cmStructureElement(dim, dim, seValues);
    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), img->
    cmGetImageType());
    int i, j, x, y, b, aux, layers = 1, auxIDimh, auxJDimw, auxIDimwx, auxJDimwY;
    if (img->cmIsRGB()) layers = 3;

    //scaning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dim2;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dim2;
            //scanning the image layers
            for (b = 0; b < layers; b++) {
                //the first maximum value is the minimum value possible
                aux = 0;
                for (x = 0; x < dim; x++) {
                    //geting a coordinate to be accessed on the image
                    auxIDimwx = auxIDimh + x;
                    //if the coordinate is smaller than 0 or bigger than the image
                    dimension
                    //the coordinate is outside of the image and cannot do the
                    calculus, so
                    //the algorithm goes to the next coordinate possible
                    if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight()))
                    continue;
                    for (y = 0; y < dim; y++) {
                        auxJDimwY = auxJDimw + y;
                        //the same problem with the coordinates outside of the image
                        if ((auxJDimwY < 0) || (auxJDimwY) >= img->cmGetWidth())
                        continue;
                        if (se->cmGetValue(x, y) == 0) continue;
                        aux += se->cmGetValue(x, y) * img->cmGetPixel(auxIDimwx,
                        auxJDimwY, b);
                    }
                }
                //allocatig the minimum value to the image coordinates
                imgResult->cmSetPixel(i, j, b, aux);
            } // layers
        } // j
    } // i
    return imgResult;

}
```

Quadro 46 – Algoritmo Simplificado da Função Morfológica de Erosão – cmErode.

```
/**
 * Erode an image with the structure element.
 * @param img The image to be eroded.
 * @param se The structure element to erode the image.
 * @return The eroded image.
 */
cmImage * cmFunctions::cmErode(cmImage * img, cmStructureElement * se) {

    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmErode: Cannot erode a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmErode: Cannot erode an image with a NULL Structure Element.";
        return NULL;
    }

    int i, j, x, y, b, min, aux, dimw, dimh;
    int auxIDimh, auxIDimwx, auxJDimw;

    //discovering the neighborhood size
    dimh = se->cmGetHeight() / 2;
    dimw = se->cmGetWidth() / 2;

    //eroding a grayscale or binary image
    int imgtype = GRAYSCALE;
    if (img->cmIsBinary()) imgtype = BINARY;
    //creating the grayscale image result
    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), imgtype);

    //scanning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dimh;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dimw;
            //the first minimun value is the maximum value possible
            min = 255;
            //scanning the structure element
            for (x = 0; x < se->cmGetHeight(); x++) {
                //geting a coordinate to be accessed on the image
                auxIDimwx = auxIDimh + x;
                //if the coordinate is smaller than 0 or bigger than the image dimension
                //the coordinate is outside of the image and cannot do the calculus, so
                //the algorithm goes to the next coordinate possible
                if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) continue;
                for (y = 0; y < se->cmGetWidth(); y++) {
                    //the same problem with the coordinates outside of the image
                    if ((auxJDimw + y < 0) || (auxJDimw + y) >= img->cmGetWidth()) continue;
                    if (se->cmGetValue(x, y) == 1) {
                        aux = img->cmGetRColor(auxIDimwx, auxJDimw + y);
                        //saving the minimum value
                        if (min > aux) min = aux;
                    }
                }
            }
            //allocatig the minimum value to the image coordinates
            imgResult->cmSetRColor(i, j, min);
        }
    }
    //returning the image result
    return imgResult;
}
```

Quadro 47 – Algoritmo Simplificado da Função Morfológica de Dilatação – cmDilate.

```
/**
 * Dilate an image with the structure element.
 * @param img The image to be dilated.
 * @param se The structure element to dilate the image.
 * @return The dilated image.
 */
cmImage * cmFunctions::cmDilate(cmImage * img, cmStructureElement * se) {

    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmDilate: Cannot dilate a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmDilate: Cannot dilate an image with a NULL
        Structure Element.";
        return NULL;
    }

    int i, j, x, y, b, max, aux, dimw, dimh;
    int auxIDimh, auxIDimwx, auxJDimw;

    //discovering the neighborhood size
    dimh = se->cmGetHeight() / 2;
    dimw = se->cmGetWidth() / 2;

    //dilating a grayscale or binary image
    int imgtype = GRAYSCALE;
    if (img->cmIsBinary()) imgtype = BINARY;

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), imgtype);

    //scaning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dimh;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dimw;
            //the first maximum value is the minimum value possible
            max = 0;
            for (x = 0; x < se->cmGetHeight(); x++) {
                //geting a coordinate to be accessed on the image
                auxIDimwx = auxIDimh + x;
                //if the coordinate is smaller than 0 or bigger than the image dimension
                //the coordinate is outside of the image and cannot do the calculus, so
                //the algorithm goes to the next coordinate possible
                if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) continue;
                for (y = 0; y < se->cmGetWidth(); y++) {
                    //the same problem with the coordinates outside of the image
                    if ((auxJDimw + y < 0) || (auxJDimw + y) >= img->cmGetWidth()) continue;
                    if (se->cmGetValue(x, y) == 1) {
                        aux = img->cmGetRColor(auxIDimwx, auxJDimw + y);
                        //saving the maximum value
                        if (max < aux) max = aux;
                    }
                }
            }
            //allocatig the minimum value to the image coordinates
            imgResult->cmSetRColor(i, j, max);
        }
    }
    //returning the image result
    return imgResult;
}
```

Quadro 48 – Algoritmo da Função Morfológica de Abertura – cmOpen.

```
/**
 * Execute the open operator in an image using a structure element and return the image result.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmOpen(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmOpen: Cannot execute the Open operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmOpen: Cannot execute the Open operation using a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmErode(img,se);
    cmStructureElement * seT = cmFunctions::cmTranspose(se);
    cmImage * imgResult = cmFunctions::cmDilate(imgEro,seT);
    delete(imgEro);
    delete(seT);
    return imgResult;
}
```

Quadro 49 – Algoritmo da Função Morfológica de Fechamento – cmClose.

```
/**
 * Execute the close operator in an image using a structure element and return the image result.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmClose(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmClose: Cannot execute the Close operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmClose: Cannot execute the Close operation using a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgDil = cmFunctions::cmDilate(img,se);
    cmStructureElement * seT = cmFunctions::cmTranspose(se);
    cmImage * imgResult = cmFunctions::cmErode(imgDil,seT);
    delete(imgDil);
    delete(seT);
    return imgResult;
}
```

Quadro 50 – Algoritmo da Função do Gradiente da Erosão – cmGradientInternal.

```
/**
 * Execute the internal gradient operator of an image using a structure element and
return the image result.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmGradientInternal(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientInternal:
        Cannot execute the Internal Gradient operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientInternal:
        Cannot execute the Internal Gradient operation using a NULL Structure
        Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmErode(img,se);
    cmImage * imgResult = cmFunctions::cmSubtractImages(img,imgEro);
    delete(imgEro);
    return imgResult;
}
```

Quadro 51 – Algoritmo da Função do Gradiente da Dilatação – cmGradientExternal.

```
/**
 * Execute the external gradient operator of an image using a structure element and
return the image result.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmGradientExternal(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientExternal:
        Cannot execute the External Gradient operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientExternal:
        Cannot execute the External Gradient operation using a NULL Structure
        Element.";
        return NULL;
    }
    cmImage * imgDil = cmFunctions::cmDilate(img,se);
    cmImage * imgResult = cmFunctions::cmSubtractImages(imgDil,img);
    delete(imgDil);
    return imgResult;
}
```

Quadro 52 – Algoritmo da Função do Gradiente Total – cmGradientTotal.

```
/**
 * Execute the total gradient operator of an image using a structure element and return the image result.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmGradientTotal(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientTotal: Cannot execute the Total Gradient operation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGradientTotal: Cannot execute the Total Gradient operation using a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmErode(img,se);
    cmImage * imgDil = cmFunctions::cmDilate(img,se);
    cmImage * imgResult = cmFunctions::cmSubtractImages(imgDil,imgEro);
    delete(imgEro);
    delete(imgDil);
    return imgResult;
    return cmFunctions::cmSubtractImages(cmFunctions::cmDilate(img, se), cmFunctions::cmErode(img, se));
}
```

Quadro 53 – Algoritmo da Função do Combinado Mínimo – cmGMin.

```
/**
 * Calculate de combined minimum (GMin) between the erosion residue and dilatation residue.
 * @param img The original image to have the GMin calculated.
 * @param se The structure element to be used int the erosion and dilatation.
 * @return The image created in GMin operation.
 */
cmImage * cmFunctions::cmGMin(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGMin: Cannot execute the GMin in a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGMin: Cannot execute the GMin in a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmGradientInternal(img, se);
    cmImage * imgDil = cmFunctions::cmGradientExternal(img, se);
    cmImage * imgResult = cmFunctions::cmMin(imgEro, imgDil);
    delete(imgEro);
    delete(imgDil);
    return imgResult;
}
```

Quadro 54 – Algoritmo da Função do Combinado Máximo – cmGMax.

```
/**
 * Calculate de combined maximum (GMax) between the erosion residue and dilatation
 residue.
 * @param img The original image to have the GMax calculated.
 * @param se The structure element to be used int the erosion and dilatation.
 * @return The image created in GMax operation.
 */
cmImage * cmFunctions::cmGMax(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGMax: Cannot execute the
        GMax in a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGMax: Cannot execute the
        GMax in a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmGradientInternal(img, se);
    cmImage * imgDil = cmFunctions::cmGradientExternal(img, se);
    cmImage * imgResult = cmFunctions::cmMax(imgEro, imgDil);
    delete(imgEro);
    delete(imgDil);
    return imgResult;
}
```

Quadro 55 – Algoritmo da Função do Combinado da Soma – cmGSum.

```
/**
 * Calculate de sum combined (GSum) between the erosion residue and dilatation
 residue.
 * @param img The original image to have the GSum calculated.
 * @param se The structure element to be used int the erosion and dilatation.
 * @return The image created in GSum operation.
 */
cmImage * cmFunctions::cmGSum(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGSum: Cannot execute the
        GSum in a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGSum: Cannot execute the
        GSum in a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgEro = cmFunctions::cmGradientInternal(img, se);
    cmImage * imgDil = cmFunctions::cmGradientExternal(img, se);
    cmImage * imgResult = cmFunctions::cmSumImages(imgEro, imgDil);
    delete(imgEro);
    delete(imgDil);
    return imgResult;
}
```

Quadro 56 – Algoritmo da Função do Combinado de Borramento Mínimo – cmGBlur.

```
/**
 * Calculate de blur combined (GBlur) between the erosion residue and dilatation
 residue.
 * @param img The original image to have the GBlur calculated.
 * @param se The structure element to be used int the erosion and dilatation.
 * @return The image created in GBlur operation.
 */
cmImage * cmFunctions::cmGBlur(cmImage * img, cmStructureElement * se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGBlur: Cannot execute
        the GBlur in a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmGBlur: Cannot execute
        the GBlur in a NULL Structure Element.";
        return NULL;
    }

    cmImage * imgBlur = cmFunctions::cmFilterAVG(img, se);
    cmImage * imgEro = cmFunctions::cmErode(imgBlur, se);
    cmImage * imgDil = cmFunctions::cmDilate(imgBlur, se);
    cmImage * imgSub1 = cmFunctions::cmSubtractImages(imgBlur,imgEro);
    delete(imgEro);
    cmImage * imgSub2 = cmFunctions::cmSubtractImages(imgDil,imgBlur);
    delete(imgDil);
    delete(imgBlur);
    cmImage * imgResult = cmMin(imgSub1,imgSub2);
    return imgResult;
}
```

Quadro 57 – Algoritmo da Função de Top-hat por Abertura – cmTophatOpen.

```
/**
 * Execute the Tophat transformation using the opening (cmOpen) operation.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmTophatOpen(cmImage* img, cmStructureElement* se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmTophatOpen: Cannot
        execute the Tophat transformation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmTophatOpen: Cannot
        execute the Tophat transformation using a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgOpen = cmFunctions::cmOpen(img,se);
    cmImage * imgResult = cmFunctions::cmSubtractImages(img,imgOpen);
    return imgResult;
}
```

Quadro 58 – Algoritmo da Função de Top-hat por Fechamento – cmTophatClose.

```
/**
 * Execute the Tophat transformation using the closing (cmClose) operation.
 * @param img The image to be processed.
 * @param se The structure element used during the process.
 * @return The image result.
 */
cmImage * cmFunctions::cmTophatClose(cmImage* img, cmStructureElement* se) {
    if (img == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmTophatClose: Cannot
        execute the Tophat transformation using a NULL image.";
        return NULL;
    }
    if (se == NULL) {
        cout << endl << "CARTOMORPH ERROR in the function cmTophatClose: Cannot
        execute the Tophat transformation using a NULL Structure Element.";
        return NULL;
    }
    cmImage * imgClose = cmFunctions::cmClose(img,se);
    cmImage * imgResult = cmFunctions::cmSubtractImages(imgClose,img);
    return imgResult;
}
```

Quadro 59 – Algoritmo da Função do Filtro Gaussiano – cmGaussianFilter.

```
/**
double cmFunctions::cmFunG(double x, double sd) {
    return ((1 / (sd * sqrt(2 * M_PI))) * exp(-1 * ((x * x) / (2 * (sd * sd)))));
}

/**
cmImage * cmFunctions::cmGaussianFilter(cmImage * img, int dim, double sd) {

    if (img == NULL) {
    if ((dim < 0) || (dim % 2 == 0)) {

    double * mat = (double*) calloc(dim*dim, sizeof (double));
    double aux, soma = 0;
    int i, j, auxi, i2, dim2 = dim / 2;

    for (i = (-1 * dim2); i <= dim2; i++) {
        auxi = (i + dim2) * dim;
        i2 = i*i;
        for (j = (-1 * dim2); j <= dim2; j++) {
            aux = cmFunctions::cmFunG(sqrt((float) (i2 + j * j)), sd);
            *(mat + auxi + (j + dim2)) = aux;
            soma += aux;
        }
    }

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), img->cmGetImageType());
    int x, y, b, layers = 1, auxIDimh, auxJDimw, auxIDimwx, auxJDimwY;

    if (img->cmIsRGB()) layers = 3;

    //scaning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dim2;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dim2;
            //scanning the image layers
            for (b = 0; b < layers; b++) {
                //the first maximum value is the minimum value possible
                aux = soma = 0;
                for (x = 0; x < dim; x++) {
                    //geting a coordinate to be accessed on the image
                    auxIDimwx = auxIDimh + x;
                    //if the coordinate is smaller than 0 or bigger than the image dimension
                    //the coordinate is outside of the image and cannot do the calculus, so
                    //the algorithm goes to the next coordinate possible
                    if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) continue;
                    for (y = 0; y < dim; y++) {
                        auxJDimwY = auxJDimw + y;
                        //the same problem with the coordinates outside of the image
                        if ((auxJDimwY < 0) || (auxJDimwY) >= img->cmGetWidth()) continue;
                        aux += *(mat + x * dim + y) * img->cmGetPixel(auxIDimwx, auxJDimwY, b);
                        soma += *(mat + x * dim + y);
                    }
                }
                //allocatig the minimum value to the image coordinates
                x = (int) (aux / soma);
                imgResult->cmSetPixel(i, j, b, x);
            } // layers
        } // j
    } // i
    return imgResult;
}
```

Quadro 60 – Algoritmo da Função do Filtro Bilateral – cmBilateralFilter.

```
/**
cmImage * cmFunctions::cmBilateralFilter(cmImage * img, int dim, double sig_s, double sig_r) {

    if (img == NULL) {
    if ((dim < 0) || (dim % 2 == 0)) {

    int i, j, auxi, i2, dim2 = dim / 2;
    double * mat = (double*) calloc(dim*dim, sizeof (double));
    double aux, Wp, sum;

    for (i = (-1 * dim2); i <= dim2; i++) {
        auxi = (i + dim2) * dim;
        i2 = i*i;
        for (j = (-1 * dim2); j <= dim2; j++) {
            *(mat + auxi + (j + dim2)) = cmFunctions::cmFunG(sqrt((float) (i2 + j * j)), sig_s);
        }
    }

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(), img->cmGetImageType());
    int x, y, b, v, v0, layers = 1, auxIDimh, auxJDimw, auxIDimwx, auxJDimwY;
    int teste = 0;
    if (img->cmIsRGB()) layers = 3;

    //scaning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        auxIDimh = i - dim2;
        for (j = 0; j < img->cmGetWidth(); j++) {
            auxJDimw = j - dim2;
            //scanning the image layers
            for (b = 0; b < layers; b++) {
                //the first maximum value is the minimum value possible
                v0 = img->cmGetPixel(i, j, b);
                Wp = sum = 0;
                for (x = 0; x < dim; x++) {
                    //geting a coordinate to be accessed on the image
                    auxIDimwx = auxIDimh + x;
                    //if the coordinate is smaller than 0 or bigger than the image dimension
                    //the coordinate is outside of the image and cannot do the calculus, so
                    //the algorithm goes to the next coordinate possible
                    if ((auxIDimwx < 0) || (auxIDimwx >= img->cmGetHeight())) continue;
                    for (y = 0; y < dim; y++) {
                        auxJDimwY = auxJDimw + y;
                        //the same problem with the coordinates outside of the image
                        if ((auxJDimwY < 0) || (auxJDimwY) >= img->cmGetWidth()) continue;

                        v = img->cmGetPixel(auxIDimwx, auxJDimwY, b);
                        aux = *(mat + x * dim + y) * cmFunctions::cmFunG(v0 - v, sig_r);
                        Wp += aux;
                        sum += aux*v;
                    }
                }
                //allocatig the minimum value to the image coordinates
                teste = (int) (sum / Wp);
                imgResult->cmSetPixel(i, j, b, teste);
            } // layers
        } // j
    } // i
    return imgResult;
}
```

Quadro 61 – Algoritmo da Função de Rotular os Alvos de uma Imagem – cmLabel.

```
/**
void cmFunctions::cmLabelAux(cmImage * img, unsigned int * mat, cmStructureElement * se, int i, int j, int label, int * count, void * pQueue, int targetColor) {

    if (img->cmGetRColor(i, j) != targetColor) return;
    //    if ((*imgR)->cmGetRColor(i, j) != 0) return;
    if (*(mat + i * img->cmGetWidth() + j) != 0) return;

    //    (*imgR)->cmSetRColor(i, j, label);
    *(mat + i * img->cmGetWidth() + j) = label;

    (*count)++;
    cmPoint pAux;

    queue<cmPoint> * myQueue = (queue<cmPoint>*) pQueue;

    int x, y, auxX, auxY;
    auxX = i - (se->cmGetHeight() / 2);
    auxY = j - (se->cmGetWidth() / 2);
    for (x = 0; x < se->cmGetHeight(); x++) {
        pAux.x = auxX + x;
        if (((pAux.x) < 0) || ((pAux.x) >= img->cmGetHeight())) continue;
        for (y = 0; y < se->cmGetWidth(); y++) {
            if (se->cmGetValue(x, y) == 0) continue;
            pAux.y = auxY + y;
            if (((pAux.y) < 0) || ((pAux.y) >= img->cmGetWidth())) continue;
            myQueue->push(pAux);
        }
    }
}

/**
unsigned int * cmFunctions::cmLabel(cmImage * img, cmStructureElement * se, int targetColor, int * labels) {

    if (img == NULL) {
    if (se == NULL) {
    if (!img->cmIsBinary()) {
    //creating the grayscale image result
    unsigned int * matLabel = ((unsigned int *) calloc(img->cmGetWidth() * img->cmGetHeight(), sizeof (unsigned int)));
    int i, j, count, label = 1;
    queue<cmPoint> * pQueue = new queue<cmPoint>;
    cmPoint pAux;

    //scanning the image
    for (i = 0; i < img->cmGetHeight(); i++) {
        for (j = 0; j < img->cmGetWidth(); j++) {
            //It doesn't label the background
            if (img->cmGetRColor(i, j) != targetColor) continue;
            count = 0;
            cmLabelAux(img, matLabel, se, i, j, label, &count, pQueue, targetColor);
            while (!pQueue->empty()) {
                pAux = pQueue->front();
                pQueue->pop();
                cmLabelAux(img, matLabel, se, pAux.x, pAux.y, label, &count, pQueue, targetColor);
            }
            if (count > 0)
                label++;
        }
    }
    if (labels != NULL)
        *labels = label;
    //returning the image result
    return matLabel;
}
```

Quadro 62 – Algoritmo das Funções de Abertura e Fechamento por Área – cmAreaOpen e cmAreaClose.

```
/**
cmImage * cmFunctions::cmAreaOC(cmImage * img, cmStructureElement * se, int
threshold, int targetColor) {

    int labels, i, j, aux;
    unsigned int * labelImg = cmFunctions::cmLabel(img, se, targetColor, &labels);
    unsigned int * aux2;

    //calculating a histogram for the labeled image
    int * histogram = (int*) calloc(labels, sizeof (int));
    for (i = 0; i < img->cmGetHeight(); i++) {
        aux2 = labelImg + i * img->cmGetWidth();
        for (j = 0; j < img->cmGetWidth(); j++) {
            *(histogram + *(aux2 + j)) += 1;
        }
    }

    //Erasing the objects smaller than the threshold
    for (i = 0; i < img->cmGetHeight(); i++) {
        for (j = 0; j < img->cmGetWidth(); j++) {
            aux = *(labelImg + i * img->cmGetWidth() + j);
            if (aux == BLACK) continue;
            if (*(histogram + aux) < threshold) {
                *(labelImg + i * img->cmGetWidth() + j) = BLACK;
            }
        }
    }

    //creating the result image
    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(),
    BINARY);
    aux = WHITE - targetColor;
    for (i = 0; i < img->cmGetHeight(); i++) {
        for (j = 0; j < img->cmGetWidth(); j++) {
            if (*(labelImg + i * img->cmGetWidth() + j) > 0) {
                imgResult->cmSetRColor(i, j, targetColor);
            } else {
                imgResult->cmSetRColor(i, j, aux);
            }
        }
    }

    delete(labelImg);
    delete(histogram);

    return imgResult;
}

/**
cmImage * cmFunctions::cmAreaOpen(cmImage * img, cmStructureElement * se, int
threshold) {
    if (img == NULL) {
    if (se == NULL) {
    if (!img->cmIsBinary()) {

    return cmAreaOC(img, se, threshold, WHITE);
}

/**
cmImage * cmFunctions::cmAreaClose(cmImage * img, cmStructureElement * se, int
threshold) {
    if (img == NULL) {
    if (se == NULL) {
    if (!img->cmIsBinary()) {

    return cmAreaOC(img, se, threshold, BLACK);

}
```

Quadro 63 – Algoritmo da Função de Afinamento – cmThinning.

```
/**
cmImage * cmFunctions::cmThinning(cmImage * img) {

    if (img == NULL) {
    if (!img->cmIsBinary()) {

    //creating the structure elements necessaries
    //Note that the value 2 is used as a "don't matter point", in other words,
    //this point is not important for the algorithm.
    int seValues[] = {0, 0, 0, 2, 1, 2, 1, 1, 1};
    cmStructureElement * se = new cmStructureElement(3, 3, seValues);

    //Creating the result and an auxiliar image as copies of the parameter image
    cmImage * imgResult = new cmImage(img);
    cmImage * imgAux = new cmImage(img);

    //defining variables of control
    int i, j, k, x, y, aux, auxI;
    bool equal, changeTotal = true;
    int w = img->cmGetWidth() - 2;
    int h = img->cmGetHeight() - 2;

    //the algorithm stop when does not have any more change in the image
    while (changeTotal) {
        changeTotal = false;

        //the structure elements must be used in 4 different rotations
        for (k = 0; k < 4; k++) {
            //Scanning all image pixels
            for (i = 0; i < h; i++) {
                for (j = 0; j < w; j++) {

                    //if the image pixel is black go to the next pixel.
                    if (imgResult->cmGetRColor(i + 1, j + 1) == BLACK) continue;

                    equal = true;
                    //scanning all structure element points
                    for (x = 0; x < 3; x++) {
                        auxI = i + x;
                        for (y = 0; y < 3; y++) {
                            aux = se->cmGetValue(x, y);
                            //ignoring points labeled as number 2 of the structure element
                            if (aux > 1) continue;
                            aux *= WHITE;
                            //checking if the the pixel of the image is different of the structure element point
                            if (aux != imgResult->cmGetRColor(auxI, j + y)) {
                                equal = false;
                                break;
                            }
                        }
                        if (!equal) break;
                    }
                    if (equal) {
                        imgAux->cmSetRColor(i + 1, j + 1, BLACK);
                        changeTotal = true;
                    }
                }
            }
            //finished to scan the image, so rotating the strutcure element for the next iteration.
            se = se->cmRotate();
            //deleting the image result and creating a copy of the auxiliar image.
            delete (imgResult);
            imgResult = new cmImage(imgAux);
        }
    }
    return imgResult;
}
```

Quadro 64 – Algoritmos das Funções de Erosão e Dilatação Condicional – cmErodeCond e cmDilateCond.

```
/**
cmImage * cmFunctions::cmErodeCond(cmImage * img, cmImage * imgcond,
cmStructureElement * se) {

    if ((img == NULL) || (imgcond == NULL)) {
    if (se == NULL) {
    if ((img->cmGetWidth() != imgcond->cmGetWidth()) || (img->cmGetHeight() !=
    imgcond->cmGetHeight())) {

    cmImage * imgResult = cmFunctions::cmMax(img, imgcond);
    cmImage * imgEro, * imgAux;
    imgAux = imgEro = NULL;

    do {
        delete(imgAux);
        delete(imgEro);
        imgEro = cmFunctions::cmErode(imgResult, se);
        imgAux = imgResult;
        imgResult = cmMax(imgEro, img);
    } while (!cmFunctions::cmIsEqual(imgAux, imgResult));
    return imgResult;

}

/**
cmImage * cmFunctions::cmDilateCond(cmImage * img, cmImage * imgcond,
cmStructureElement * se) {

    if ((img == NULL) || (imgcond == NULL)) {
    if (se == NULL) {
    if ((img->cmGetWidth() != imgcond->cmGetWidth()) || (img->cmGetHeight() !=
    imgcond->cmGetHeight())) {

    cmImage * imgResult = cmFunctions::cmMin(img, imgcond);
    cmImage * imgDil, * imgAux;
    imgAux = imgDil = NULL;

    do {
        delete(imgAux);
        delete(imgDil);
        imgDil = cmFunctions::cmDilate(imgResult, se);
        imgAux = imgResult;
        imgResult = cmMin(imgDil, img);
    } while (!cmFunctions::cmIsEqual(imgAux, imgResult));
    return imgResult;
}
```

Quadro 65 – Algoritmo da Função de Crescimento por Região – cmGrowthRegion.

```
/**
cmImage * cmFunctions::cmGrowthRegion(cmImage * img, cmImage * imgSamples) {

    if (img == NULL) {
    if (imgSamples == NULL) {
    if (img->cmGetImageType() != GRAYSCALE) {
    if (imgSamples->cmGetImageType() != BINARY) {

    int * histogram = img->cmGetHistogramPartial(imgSamples);

    int i, max, min;
    max = 0; min = 255;

    for (i = 0; i <= 255; i++) {
        if (*(histogram + i) != 0) {
            min = i;
            break;
        }
    }
    for (i = 255; i > 0; i--) {

    if (max < min) {

    int qtdLabel, j;
    cmStructureElement * se = new cmStructureElement(3, 3, SE_BOX);
    cmImage * imgLabels = cmFunctions::cmLabelImg(imgSamples, se, WHITE, &qtdLabel
    );
    queue<cmPoint> pQueue;
    cmPoint aux, aux2;

    cmImage * imgResult = new cmImage(img->cmGetWidth(), img->cmGetHeight(),
    BINARY);

    //finding a point for all samples that will be used as start points
    for (i = 1; i < qtdLabel; i++) {
        cmFindPoint(imgLabels, i, &aux);
        pQueue.push(aux);
        imgResult->cmSetRColor(aux.x, aux.y, WHITE);
    }

    //growing region methodology
    while (!pQueue.empty()) {
        aux = pQueue.front();
        pQueue.pop();
        //The "Structure Element"
        for (i = -1; i <= 1; i++) {
            aux2.x = aux.x + i;
            //Out of the image boundaries
            if (aux2.x < 0) continue;
            if (aux2.x > img->cmGetHeight()) continue;
            //The "Structure Element"
            for (j = -1; j <= 1; j++) {
                //It is not necessary in the center point because it is already
                the aux point.
                if (i == 0 && j == 0) continue;
                aux2.y = aux.y + j;
                //Out of the image boundaries
                if (aux2.y < 0) continue;
                if (aux2.y > img->cmGetWidth()) continue;
                //The point was already done
                if (imgResult->cmGetRColor(aux2.x, aux2.y) == WHITE) continue;
                //cheking if the point belong to the interest feature
                if (cmBelong(img, aux2.x, aux2.y, max, min)) {
                    imgResult->cmSetRColor(aux2.x, aux2.y, WHITE);
                    //including the point in the Queue
                    pQueue.push(aux2);
                }
            }
        }
    }
    return imgResult;
}
```

Quadro 66 – Algoritmo da Função de Detecção Semiautomática de Alvos – cmFeatureDetection.

```
/**
 * A supervised feature detection methodology. This function try to detect the
 * interest feature present in the imgOriginal parameter using the imgSamples
 * parameter image as samples of the interest feature.
 * @param imgOriginal A GRAYSCALE image containing the original scene.
 * @param imgSamples A BINARY image where the white pixels must represent the interest feature.
 * @return A BINARY image with the detected feature in white color.
 */
cmImage * cmFunctions::cmFeatureDetection(cmImage * imgOriginal, cmImage * imgSamples ) {

    if (imgOriginal == NULL) {
    if (imgSamples == NULL) {
    if (!imgSamples->cmIsBinary()) {

    cmStructureElement * se = new cmStructureElement(3, 3, SE_BOX);
    cmImage * imgExt = cmFunctions::cmGrowthRegion(imgOriginal, imgSamples);

    cmImage* imgClose = cmFunctions::cmClose(imgExt, se);
    delete(imgExt);
    delete(se);

    se = new cmStructureElement(3, 3, SE_CROSS);
    cmImage * imgAClose = cmFunctions::cmAreaClose(imgClose, se, 1500);
    delete(imgClose);
    delete(se);
    return imgAClose;

}
```

GUILHERME PINA CARDIM - Possui graduação em Ciência da Computação pela Universidade Estadual Paulista Júlio de Mesquita Filho, mestrado e doutorado em Ciências Cartográficas pela Universidade Estadual Paulista Júlio de Mesquita Filho, sendo o último realizado em convênio cotutela com a Universidad de Alcalá (UAH) obtendo-se também o doutorado em Eletrônica: Sistemas Eletrônicos Avançados. Sistemas Inteligentes. Docente na Universidade Estadual Paulista Júlio de Mesquita Filho com experiência em pesquisa na área de Geociências, com ênfase em Sensoriamento Remoto, atuando principalmente nos seguintes temas: extração de feições cartográficas, detecção de características de interesse, processamento digital de imagens, controle de qualidade.

Processamento Morfológico

DE IMAGENS DE SENSORIAMENTO REMOTO

www.atenaeditora.com.br

contato@atenaeditora.com.br

@atenaeditora

www.facebook.com/atenaeditora.com.br

Atena Editora

Ano 2024

Processamento Morfológico

DE IMAGENS DE SENSORIAMENTO REMOTO

www.atenaeditora.com.br
contato@atenaeditora.com.br
@atenaeditora
www.facebook.com/atenaeditora.com.br

Atena Editora
Ano 2024